DTIC FILE COPY

AD-A230 602

SQL/NF TRANSLATOR FOR THE TRITON NESTED RELATIONAL DATABASE SYSTEM

THESIS

Craig William Schnepf
Captain, USAF

AFIT/GCE/ENG/90D-05

Best Available Copy

DTIC
S ELECTE D
JAN 08 1991
E

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

91 1 3 068

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

①

AFIT/GCE/ENG/90D-05

SQL/NF TRANSLATOR FOR THE TRITON NESTED RELATIONAL DATABASE SYSTEM

THESIS

Craig William Schnepf
Captain, USAF

AFIT/GCE/ENG/90D-05

DTIC
ELECTE
JAN 08 1991
S E D

Approved for public release; distribution unlimited

AFIT/GCE/ENG/90D-05

SQL/NF TRANSLATOR FOR THE TRITON NESTED RELATIONAL DATABASE SYSTEM

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Engineering)

Craig William Schnepf, A.A.S, B.S.E.E.

Captain, USAF

December, 1990

Accession For	
NTIS GRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Approved for public release; distribution unlimited

QUALITY INSPECTED 4

Acknowledgments

I am very grateful to all the people who helped and supported me throughout the thesis project. First, my advisor, Major Mark Roth, whose advice and guidance was the keystone to my understanding and implementing the thesis project. I also want to thank my instructors at AFIT for providing me with the background in software engineering that was necessary to complete a task of this size and complexity. I especially want to thank my wonderful wife Kathy whose patience and understanding made it possible for me to work the many long hours and to remind me of the other things important to life.

Craig William Schnepf

Table of Contents

List of Figures

Figure Page

AFIT/GCE/ENG/90D-05

Abstract

The problem addressed in this thesis concerns the design and implementation of a high level data base query language translator based on the nested relational data model. The objective of the model is to increase the performance of the relational model by modeling real-world objects in the problem domain into nested relations. The translator is designed within the EXODUS extensible architectural framework for building application-specific database systems. The SQL/NF query language used for the nested relational model is an extension of the popular relational model query language SQL. The query language is translated into a nested relational algebra (Colby algebra) in the form of a query tree structure. A large amount of theory exists for the nested relational model, however, very little information on the implementation of a high level query language for the model is available. This thesis effort provided the front end to a proto-type nested relational data base management system (Triton) using the EXODUS tool kit.

In Triton, EXODUS is used to implement the first stages of the data base management system. These stages include the *parser* and the *catalog manager*. Since the EXODUS tool kit does not provide a parser development tool, we chose to use the popular UNIX tools "LEX" and "YACC" to parse the SQL/NF query statements and execute the C programs necessary for creating and maintaining a data dictionary, generating a SQL/NF query tree, and translating the SQL/NF query tree into a Colby algebra query tree. The Colby algebra tree is available for the next stage of the EXODUS architecture, the *query optimizer*. (KR)

SQL/NF TRANSLATOR

FOR THE

TRITON NESTED RELATIONAL DATABASE SYSTEM

I. *Introduction*

The storage and retrieval of information by computers plays an increasingly important role in our daily lives. Banking transactions, inventory management, personnel record keeping, and payroll processing are only a few of the many traditional applications which require the efficient manipulation of large amounts of data. The manipulation of these large amounts of data is accomplished through the use of some type of database management system (DBMS) The desire to utilize database systems for what are commonly known as non-traditional applications is on the rise. Some of these non-traditional applications include computer-aided design (CAD), computer-aided manufacturing (CAM), computer-aided engineering (CAE), and audio/video data. In this thesis we address a database model which can meet the requirements of these applications along with a query language to manipulate the database.

The purpose of this thesis project is to design and implement a translator for SQL/NF that will parse a subset of SQL/NF query statements and produce a QUERY tree structure that will be compatible with the recursive algebra for nested relations developed by Latha S. Colby (4). This algebra for nested relations (Colby Algebra) was chosen as the nested algebra for the translator because it allows tuples at all levels of nesting in a nested relation to be accessed and modified without any special navigational operators and without having to flatten the nested relation. The SQL/NF translator will be the "front end" to the Triton nested relational database management system, using the EXODUS tool kit.

1.1 Background

The relational database model, introduced by Codd (3), is the most prevalent design used in commercial database systems today. The relational model is best described as a collection of relations, which are represented by tables with common properties or attributes. The tables are arranged so that the columns contain the attributes of the relation, and each row in the table contains an instance of the relation, which is known as a tuple. Operations are provided by the database system to modify or retrieve information from the relations in the database. These operations are based on relational algebra and calculus, and provide the foundation for on-going work in relational database models.

The traditional relational model requires that all values in a relation be atomic in nature. That is, each attribute of each tuple in the relation must be a single entity such as a number or a character string. When a relation meets this requirement it is said to be in first-normal-form (1NF).

The theory of nested relational databases was motivated by the observation that quite often in a relational database system it is desirable to store a *set* of values for an attribute rather than a single value. In a nested relation, attributes can be relation-valued as well as atomic-valued. A relation which occurs as the value of an attribute in a tuple of another relation is said to be nested. Therefore the nested relational model allows the database to represent complex information in a format that more closely resembles the real world.

1.2 Nature of the Problem

The nested relational database model is a relatively new concept that provides for the storage of information in a database that does not fit the traditional relational database format.

"There is a growing interest in abandoning the first-normal-form assumption on which the relational database model is based. This interest has developed from the desire to extend the applicability of the relational model beyond traditional data-processing applications."(13:99)

The nested relational model extends the capability of the relational model for non-traditional data to be stored in a DBMS. A formal implementation scheme is needed to further explore the applications for and capabilities of nested relations.

SQL (Structured Query Language) is the relational database language pioneered in the IBM System R project and subsequently adopted by IBM and others as the basis for numerous commercial implementations and as the base for several extended research prototypes (see for example (15)). SQL/NF, designed at the University of Texas at Austin (13), is an extension of SQL designed to operate on nested relations. The primary objective in designing SQL/NF was to provide a language that could be used to query nested relational databases. SQL/NF was designed to enhance SQL's capabilities by modifying all the SQL operators in order to operate on nested relations. In addition to all the operators present in SQL, SQL/NF identifies two more - the "nest" and "unnest" operators, which are necessary to convert flat relations (1NF) to equivalent nested relations and vice versa.

SQL is the current industry standard for relational database systems, and SQL/NF extends this standard so that it may also work with nested relations. Therefore a complete system which extends the capabilities of SQL to operate on nested relations is the goal of a prototype nested relational database management system being developed at AFIT, known as Triton.

1.3 Scope

The SQL/NF translator is able to parse all the SQL/NF clauses as defined by the BNF in Appendix C. Implementation of the query statements is limited to the clauses required for the creation and deletion of items in the data dictionary and the clauses that will directly translate into the appropriate algebraic operations of: *select, project,* and *cartesian-product.* In addition it will be able to parse the SQL/NF operators of: *nest* and *unnest.* All these operations will be stored in a "Query tree" structure that will represent the Colby algebra (4) translation of the input query. This query tree will be in such a format to be easily used as the input to the query optimizer stage of Triton using the EXODUS tool kit.

1.4 Approach/Methodology

The primary goal in this thesis effort has been to gain an extensive knowledge of nested relational data base theory, in addition to the SQL/NF query language and use of the EXODUS tool kit. The approach we took was to first design and implement a parser using the LEX and YACC compiler tools of Unix, then design and implement the E code[1] for the catalog manager, then design and implement the C code for an SQL/NF query tree structure, and finally design and implement the C code that translates the SQL/NF query tree into a query tree structure for the Colby algebra.

1.5 Organization

Chapter 2 begins with a background discussion of the nested relational database model, followed by an overview of some of the nested relational algebra's that have been proposed, and some work closely related to this thesis. Then an overview of the EXODUS database system project as related to this thesis effort is presented. Chapter 3 begins with an in depth discussion of the nested relational algebra presented by Latha S. Colby (4) , followed by a description of the SQL/NF query language presented in (13). Chapter 4 describes the design and implementation of the translator using the EXODUS tool kit. Finally, conclusions about this thesis effort are presented, in Chapter 5, with recommendations for future work.

[1]A persistent programming language provided as part of the EXODUS tool kit. (See Section 2.5)

II. Summary of Current Knowledge

2.1 Introduction

We begin this chapter by giving a brief introduction to the concept of nested relations as compared to standard relations (those that meet the criteria for "first normal form" or 1NF). Next, we provide an overview of the development of the Nested Relational Database Model (also known as "non-first normal form" or NF^2). This we follow with a brief summary of some of the algebra's that have been developed for use in association with the Nested Relational Model (NRM). Next, we provide an overview of some related work for an SQL/NF parser. Finally, we discuss the EXODUS(2) extensible database system project.

2.2 The Nested Relational Database Model

2.2.1 First Normal Form vs NRM. A relational database is normally described as a collection of "tables" which correspond to instances of the various relations of the database (8). In a given table the rows correspond to tuples of the relation depicted, and the columns correspond to attributes of the relation. The attributes which form a particular table, or relation, compose the scheme of that relation. This scheme is usually thought of as the pattern which is followed when particular attribute (and hence tuple) values are assigned. In addition, each attribute has an associated domain from which individual values must be chosen. Thus an entry in the location row_i, $column_j$ represents the value of the ith tuple from the domain of the jth attribute.

The traditional relational model introduced by Codd (3) in 1970 requires that all values in a relation be atomic. That is, each attribute of each tuple in the table must be a single entity such as a number, or a character string. When this requirement is met the relation is said to be in First Normal Form (1NF). Although this model is sufficient for representing objects that have simple domains, complex objects cannot be represented easily. Normalization in the relational model causes a lot of fragmentation in the representation of objects. Information about objects and their relationships can be scattered over several different tables. This in turn causes queries to the database to be slow and complicated since excessive joins have to be performed among the various tables in the database (4).

The nested relational model, also known as Non-First Normal Form, is an extension of the traditional relational model without the first normal form restrictions. Without this restriction the attributes of a relation can have non-atomic values. This corresponds to allowing for unnormalized relations. Makinouchi (9) introduced this concept by suggesting that the 1NF assumption be relaxed so that the attributes can be "set-valued".

COMPANY

dno	dname	loc	ename
dno1	dname1	loc1	ename1
dno2	dname2	loc2	ename2
dno1	dname1	loc1	ename3
dno2	dname2	loc2	ename4

Figure 2.1. 1NF Database Table

In order to illustrate these concepts let us consider the simple example shown in Figure 2.1. The example is a 1NF (flat) database scheme for some company. The database consists of a table with atomic attributes of dept number (dno), dept name (dname), location (loc), and employee names (ename). If we recognize that the relation contains a "set" of employee's by dept number , then the values for the ename attribute can be nested in a more compact nested relation as shown by the scheme in Figure 2.2. The relation COMPANY now consists of the atomic attributes dno, dname, loc, and the complex attribute emps.

COMPANY

dno	dname	loc	emps
dno1	dname1	loc1	{ ename1, ename3 }
dno2	dname2	loc2	{ ename2, ename4 }

Figure 2.2. NF^2 Database Table

2.2.2 Operators for Single Attribute Nested Relations. The nested relational model can be considered a superset of the traditional model, in as much as retaining the same operations. However, most research efforts have pointed out the need for determining how one might create unnormalized (NRM) relations from normalized (1NF) relations, and what additional operations should be defined for nested relations. These requirements were addressed in 1982 by Jaeschke and Schek(7).

2.2.2.1 Nest and Unnest Operators. The first requirement addressed by Jaeschke and Schek was the definition of the "nest" operator, which transfers a flat relation into a nested relation. This nest operator works by examining the value of a particular attribute within all tuples of the relation and partitioning the tuples along the remaining attributes (8). That is, it forms a single tuple with a new attribute name in place of the set of values in the tuples being grouped together. Using the previous company database example, the nest operator will transform the 1NF table in Figure 2.1 into the nested relational table of Figure 2.2. This would be done by applying the nest operator to the "ename" attribute in Figure 2.1, compressing the associated values of the attribute into the company table of Figure 2.2, producing a new complex attribute of "emps".

It is important to note that this definition of the Nest operator only allows nesting on a single attribute at a time. This definition also does not cover the nesting of attributes within already set-valued attributes.

The second requirement addressed by Jaeschke and Schek was the definition of the "unnest" operator, which performs the inverse of the nest operator. That is, unnest will flatten out the relation created by the nest operator back into a 1NF relation. This unnesting is performed by combining each element of a nested attribute with repeated occurrences of the associated partition of unnested attributes (8). Again, using the previous company database example, the unnest operator would transform the nested relational table of Figure 2.2 into the flat relational table of Figure 2.1. This would be done by applying the unnest operator to the "emps" attribute, thus expanding the table to its original 1NF form.

2.2.2.2 Relational Algebra Operators. Jaeschke and Schek also briefly addressed the possibility of applying the normal relational algebra operators to the nested relations. They said that "union, difference, projection. and the cartesian product can be defined for all relations, so they also apply on [nested] relations." Therefore, allowing the comparison of set-valued attributes via the set comparison symbols ($\subset, \supset, \subseteq, \supseteq, =$) also "extends" selection to nested relations (8).

The last item Jaeschke and Schek defines are two methods of performing a natural join of nested relations. The first method can be considered an "extension" of the "normal" natural join (8). The second method is called the "intersection join". These methods will be further explained in Chapter 3, as they are defined for the nested relational model used by the Colby Algebra.

2.2.3 Operators for Multi-Attribute, Multi-Level Nested Relations. Expansion of the work done by Jaeschke and Schek to allow for multiple attribute nesting and multiple levels was accomplished by the efforts of Fischer and Thomas (17). They extended the definitions of the relational algebra operators in view of increasing the scope for nested relations. This included providing results concerning properties associated with the interaction of nest, unnest, and the relational algebra operators.

The nest operator of the extended definition allows each element of a set associated with a given partition to in turn be a set of attributes instead of merely a single attribute. This allows for nesting of more than one attribute at a time and for multiple levels of nesting. These unnormalized relations permit components of tuples to now also have relations as attributes, forming subrelations of the relation. The nested relational model allows users to view the database in a way that more closely represents a real world concept of complex objects. The associated relations can now be represented as a whole entity in a single nested relation instead of being distributed over several different flat relations.

The unnest operator of the extended definition repeatedly combines each element of a nested attribute with its associated partition of remaining attributes so as to "flatten" the overall structure back to its 1NF form (8). These developments will be incorporated in the discussion of the definition for the nested relational model as it is used with the Colby Algebra in the next section.

Now we have seen that although nested relations can consist of relations and atomic attributes, the basic relational structure is still the primary building block. However, the nested relational data model, to form its very structure, must provide a means to construct nested relations in addition to using the basic relational operators (10). Before proceeding to discuss these operators further, an example is in order.

DEPT

dno	dname	loc

EMP

eno	ename	dno	sal

CHILDREN

eno	cname	dob

Figure 2.3. 1NF Database Headers

A nested relational structure can be created from the three flat relations in Figure 2.3 to form the NRM relational structure of Figure 2.4. The EMP relation can be inserted in the DEPT relation as a relation-valued attribute by their common attribute of dno, and the CHILDREN relation can be inserted in the EMP relation as a relation-valued attribute by their common attribute of eno. These now form the new NRM relation of COMPANY with a relation-valued attribute of Emp and atomic attributes of dno, dname, and loc. The subrelation Emp has a relation-valued attribute of Children and atomic attributes of eno, ename, and sal. Finally, the subrelation Children has only atomic valued attributes of cname and dob. In this case, two levels of nesting have occurred within the COMPANY relation.

COMPANY

dno	dname	loc	Emp				
			eno	ename	sal	Children	
						cname	dob

Figure 2.4. Nested Database Table header

2.3 *Nested Relational Algebras*

The following sections provide a summary of some of the work currently going on in the realm of algebras for the nested relational model and a version of a translator for SQL/NF.

2.3.1 Thomas and Fischer. The algebra introduced by Thomas and Fischer (17) is simply an extension of the operators for the relational model. The definitions for the relational model operators also apply for this algebra. Thomas and Fischer added the additional operators of "nest" and "unnest" to manipulate the structure of the data being stored. However, in order for any other operation to be performed, such as "select" or "project", the subrelation must be unnested first. Then the normal operations are applied to the result. The unnesting is required because the operators can only act upon the attributes at the top most level in a relation. Once an operation is complete, the results may have to be transformed back to the original form using the nest operator. Therefore, this algebra merely stores the data according to the nested relational model, but any manipulation of the data is done at the 1NF level.

2.3.2 A Recursive Algebra for Nested Relations. The algebra introduced by Colby (4) allows subrelations at any level within a relation to be accessed and manipulated without the need for a "nest" or "unnest" operation. In this algebra, the traditional relational operators are extended with recursive definitions so that they can be applied not only to relations but also to subrelations of the relation. The subrelations are accessed using a "recursive" list of attributes that identify the path over which the extended relational operator(s) will perform their operations. The list is called recursive because a *single* element of the list can be made up of another list of attributes and so on. The main advantage of this method is that operations can be done without restructuring the relation in order to extract data that has been nested at different levels within the relation.

2.3.3 Schek and Scholl. The algebra introduced by Schek and Scholl (14) allows subrelations at any level within a relation to be accessed without having to perform a nest or unnest operation. A special operator is identified as a "navigator" to access the nested subrelations. The navigator Schek and Scholl chose to use is the projection operator (π). This navigator requires that a query performed on a subrelation is accomplished by a series of projects until the subrelation is reached, followed by the desired operation. In addition to this sometimes awkward requirement, renaming the results of the projections is necessary in a number of situations.

2.3.4 Deshpande and Larson. The algebra proposed by Deshpande and Larson (5) allows subrelations at any level within a relation to be accessed without having to perform a nest or unnest operation. Like Schek and Scholl (14), Deshpande and Larson chose to use a "navigator" type operator method to access the nested subrelations. The navigation is performed via an extended selection operator and a subrelation constructor. Their definition for the selection operator provides the ability to access any nested subrelation. This access is done using the subrelation constructor which creates new subrelations while traversing the lower levels in the relation. One added difficulty with this method is that it requires introducing new relations, and subsequently the need for the user to provide a name for this new relation in order for the selection operator to function properly.

2.4 Some Related Work

The SQL/NF translator designed by Ramakrishnan (11) translates SQL/NF queries extended with "role joins" to a nested relational algebra in three stages which are termed:

1. query transformation.
2. pre-processing.
3. meaning evaluation.

The first stage transforms the SQL/NF query statements into an intermediate form using the UNIX tools LEX and YACC. The second stage further reduce the intermediate form generated in the first stage using set-theoretic transformations. The final stage transforms the reduced intermediate form into a nested relational algebra. This algebra is similar to the traditional relational algebra introduced by Codd (3) except that a relation valued attribute may occur in any place an atomic valued attribute can occur. In addition another operator is defined called "functional evaluation", this evaluator provides for operations on nested expressions.

The result of the implementation of Ramakrishnan's translator is in the form of what is known as a "Query Tree." This query tree is a data structure representation of the algebraic translation of the SQL/NF query. One drawback of this translator is that additional software is required to produce the desired results of the original SQL/NF query. This additional software must accept the query tree data structure as input and perform the designated algebraic operations in the query tree.

2.5 EXODUS

EXODUS (2) is an extensible database system project that is addressing data management problems posed by a variety of challenging new applications. The EXODUS project is being developed at the University of Wisconsin The goal of the project is to facilitate the fast development of high-performance, application-specific database systems. EXODUS provides an architectural framework (see Figure 2.5) for building application-specific database systems. This architecture provides powerful tools to help automate the generation of application-specific database systems, including a rule-based query optimizer generator and a persistent programming language, and libraries of generic software components that are likely to be useful for many applications. The system compiler, used for the compilation of the system software, is based on the E programming language (12), an extension of the object-oriented programming language C++(16). The two stages of the EXODUS architecture that are addressed and implemented by this thesis project are the *Parser* and *Catalog Manager.*

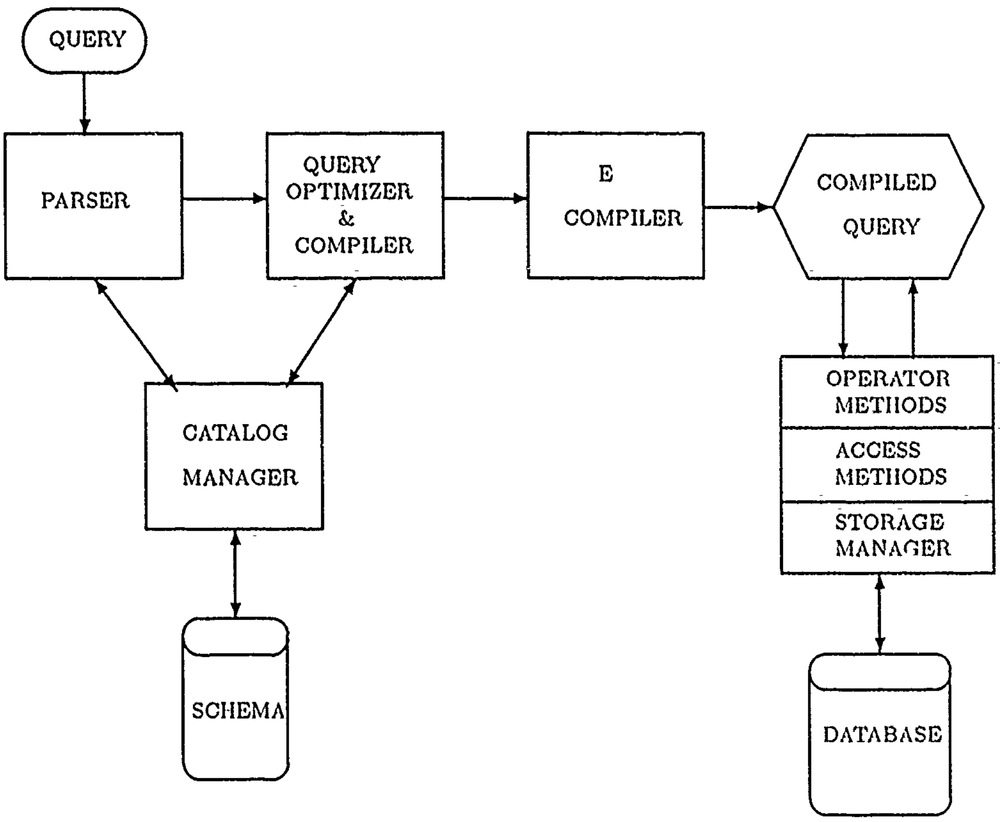

Figure 2.5. EXODUS architecture

2.5.1 Parser. The main purpose of the parser is to provide a way to link the user with the database system through a high level query language such as SQL. The EXODUS tool kit does not provide a parser development tool, so an alternative had to be found. The alternative we chose to use is the popular UNIX tools of YACC (Yet Another Compiler Compiler) and LEX (Lexical Analyzer). These tools have been proven successful in the associated projects by both

Ramakrishnan (11) and Mankus (10).

2.5.1.1 YACC program. YACC is a general tool for describing the input to a computer program. The YACC user specifies the structure of the input together with the code to be invoked as each structure is recognized. YACC turns such a specification into a subroutine called *yyparse* that handles the input process (6). The general format for a YACC program is described in Appendix A.

2.5.1.2 LEX-a lexical analyzer. LEX is a program generator designed for lexical processing of character input streams. It accepts a high-level specification for character string matching, and produces a program in a general purpose language that recognizes regular expressions. The program generated by lex is called *yylex* (6).

LEX generates lexical analyzers in a manner analogous to the way YACC creates parsers. The user inputs a specification of the lexical rules of a language using regular expressions together with fragments of C to be executed when a matching string is found. The general format for a LEX program is described in Appendix A.

2.5.1.3 SQL/NF translator parser implementation. LEX supp the ability to scan and identify the words associated with the high level language of SQL/NF and convert them into a predefined set of tokens. Once a token is returned from the scanner, the parser (YACC) matches the token to a predefined set of grammar rules. Within the grammar rules, statements in the form of C code procedure calls are executed to direct the database system to carry out some specific operation. The operations performed will generate input to and manipulate the catalog manager in addition to building the query tree.

2.5.2 Catalog manager. The catalog manager is the mechanism responsible for ensuring proper organization of relational tables within the database. Whenever a new database is created or a current one opened, all references to any of the relations and their associated attributes are handled by the mechanisms implemented by the catalog manager. The manager should also provide for persistence of the database information after termination of the database program. The EXODUS tool kit provides a generic catalog manager that can be used to provide for these requirements. Information required by the catalog manager to load its tables will be obtained through the scanning and parsing of the data definition arguments of the SQL/NF commands. Once the information is passed from the scanner, through the parser, and to the catalog manager, the data must be input into its tables in a particular format defined within the Data Dictionary. The information generated by the parser and stored in the Data Dictionary consists of two tables. The first table contains the definitions of each relational scheme and the attributes, both relation valued and atomic valued, associated with each scheme. The definitions of the relational tables are stored in the second table of the Data Dictionary along with their associated scheme identifiers. The composition of the Data Dictionary is identical to the one used by Mankus(10) and is described in Section 3.4 of his thesis.

III. Colby Algebra & SQL/NF

3.1 Colby Algebra

The nested relational algebra developed by Latha S. Colby at Indiana University (4) is equivalent in expressive power to the nested relational algebra of Thomas and Fischer (17) discussed in the previous chapter. The Colby algebra, however, allows queries to be expressed more naturally and succinctly without the need for restructuring the relation being operated upon. In the next section we will define the Nested Relational Model according to Colby (4). Followed by a description of each of Colby's NRM operators.

3.1.1 Definitions for the Nested Relational Model.

Let A be the universal set of attribute names and relation scheme names. A relation scheme of a nested relation is of the form R(S) where R $\in A$ is the relation scheme name and S is a list of the form $(A_1, A_2, \ldots, A_n)$ where each A_i is either an atomic attribute or a relation scheme of a subrelation. If A_i is a relation scheme of the form $R_i(S_i)$, then R_i, the name of the scheme, is called a relation-valued attribute of R.

For each atomic attribute A_i in A, let D_i be the corresponding domain of values. An instance r of a relation scheme R(S), where $S = (A_1, A_2, \ldots, A_n)$, is a set of ordered n-tuples of the form $(a_1, a_2, \ldots, a_n)$ such that:

1. if A_i is an atomic attribute, then $a_i \in D_i$.
2. if A_i is a relation scheme, then a_i is an instance of A_i.

An instance of a relation scheme is also referred to as a (nested) relation.

Let R(S) be a relation scheme. **Attr(R)** is the set of all (atomic and relation-valued) attribute names in S. **RAttr(R)** is the set of all relation- valued attributes in S. **FAttr(R)** is the set of all flat or atomic attributes in S. **deg(R)** is the number of attributes in S. Henceforth, when we refer to a relation scheme, we will refer to it by its name alone (e.g., R instead of R(S)). Let r be an instance of R and let $t \in r$ (a tuple in relation r). If $A \in Attr(R)$ then t(A) is the value of t in the column corresponding to A. If $B \subseteq Attr(R)$ then $t[B] = t[A_1]t[A_2]\ldots t[A_m]$ where $A_i \in B$ $(1 \leq i \leq m)$. Let c be a condition on R if:

1. c is NULL;
2. $c = a\Theta b$ where,
   (a) a is an atomic attribute of R and b is an atomic attribute or an atomic value, a and b have compatible domains and $\Theta \in \{<, >, \leq, \geq, =, \neq\}$.

   (b) a is a relation-valued attribute of R and b is a relation-valued attribute of R or an instance of the domain of a and $\Theta \in \{\subset, \supset, \subseteq, \supseteq, =, \neq\}$.

   (c) b is a relation-valued attribute of R and a is a tuple in some instance of b and $\Theta \in \{\in, \ni\}$.
3. c1 and c2 are two conditions on R and $c = c1 \wedge c2$ or $c = c1 \vee c2$ or $c = \neg c1$.

If t is a tuple in some relation $r \in R$, then:

1. If $c = \emptyset$ then c(t) = [illegible]
2. If $c = a\Theta b$ then $c(t)$ [illegible]
   (a) $t[a]\Theta t[b]$ [illegible] and b are both attributes.

   (b) $a\Theta t[b]$ if only b is an attribute.

   (c) $t[a]\Theta b$ if only a is an attribute.
3. $c(t) = c1(t) \wedge c2(t)$, $c1(t) \vee c2(t)$ and $\neg c1(t)$ when $c = c1 \wedge c2$, $c = c1 \vee c2$ and $c = \neg c1$ respectively.

In the example nested relation "Company" shown in Figure 2.4:

1. R(dno, dname, loc, Emp(eno, ename, sal, Children(cname, dob))) is the scheme of the relation.
2. Attr(R) = { dno, dname, loc, Emp }
3. FAttr(R) = { dno, dname, loc }
4. RAttr(R) = { Emp }

3.1.2 The Nested Relational Recursive Algebra. The main objective of the recursive algebra is to provide a way to access and manipulate data at all levels of a nested relation. The advantage of this is that it can be done without restructuring the relation in order to extract data that is nested at different levels within the relation. The operators of this algebra are defined in an extended form (recursively) so that they can perform their operations on all sublevels of the relation without having to use a special operator as a "navigator" to search through the relation. (The "navigator" operator is further explained in section 2.3.3.) This also reduces the requirement for the number of nest and unnest operations that would have to be performed without the recursive ability. In the following sections we briefly describe the operators as defined by Colby. A list of the formats and the formal definitions for all the operators can be found in Appendix B.

3.1.2.1 Selection (σ). The selection operator can operate on attributes at any level of the relation, even when the selection condition is not at the outermost level. This is done without having to flatten the nested relation first. The select list is the key to the recursive nature of this operator. Each subrelation (R_i) is tested to see if tuples exist so that the condition on R_i is met. This is done for all subrelations until the bottom most subrelation is tested and the condition on Ri is met. If every condition is not met (ie., an empty set is returned for one of the subrelations) then an empty set is returned for the operation.

The selection operator requires three elements to select tuples from a nested relation. First, the source relation name. The second element is a select list which permits the operator to reach

COMPANY

dno	dname	loc	Emp				
			eno	ename	sal	Children	
						cname	dob
001	Eng	Bldg3	111	Smith	12000	Jane	May
						Dave	April
			121	Jones	12500	Bob	Oct
						Sue	Jan
002	Mkt	Bldg2	222	Adams	18500	Mary	Dec
						Mark	March
						Rick	June
			245	Carter	18000	Carol	Aug
						Mike	Oct
			263	Davis	20000	Bill	Feb

Figure 3.1. Nested Database Table

any attribute, atomic or relation valued, and permits the operator to descend to any depth of the relation. The last element is a set of conditions or predicates which can be specified for the top-level atomic attributes or on any of the lower nested relations. The generic format of the selection operator is as follows:

$$\sigma(Relation_{Condition}(Select\ list))$$

As an example, to select those tuples from the COMPANY relation (see Figure 3.1) in which the department name is MKT and the employee makes more than 18000 will require the following query:

$$\sigma(COMPANY_{dname=\text{``}MKT\text{''}}(Emp_{sal>18000})).$$

The condition, dname = "MKT", on the top level atomic attribute, dname, always follows the source relation name, COMPANY, while the lower level condition, sal > 18000, is specified

dno	dname	loc	Emp				
			eno	ename	sal	Children	
						cname	dob
002	Mkt	Bldg2	222	Adams	18500	Mary	Dec
						Mark	March
						Rick	June
			263	Davis	20000	Bill	Feb

Figure 3.2. *SELECT* operator results

within the select list after the subrelation name, Emp. The resultant set of tuples are shown in Figure 3.2.

3.1.2.2 Projection (π). The projection operator can operate on attributes at any level of the relation, even when the attribute(s) being projected are not at the outermost level. This is done, like the selection operator, without having to flatten the nested relation first. The key to the recursive nature of this operator is the project list. The attributes being projected are identified by R_i which can be either atomic or relation-valued. If R_i is relation-valued then Li identifies the subrelations and/or atomic attributes of the subrelation R_i that are to be projected, and so forth. This will produce a subset of the database with the header identified by the project list.

The projection operator requires two elements to project out the requested attributes from a nested relation. First, a project list permits the operator to reach any attribute, atomic or relation valued, and project out the contents of a single attribute column or an entire subrelation. It also permits the operator to descend to any depth of the relation. The second element is the source relation name. The generic format of the projection operator is as follows:

$$\pi((Project\ List)\ Relation).$$

As an example, to project the attributes of ename and cname from the nested relation COMPANY (see Figure 3.1) the following query would be made:

Emp	
ename	Children
	cname
Smith	Jane
	Dave
Jones	Bob
	Sue
Adams	Mary
	Mark
	Rick
Carter	Carol
	Mike
Davis	Bill

Figure 3.3. *PROJECT* operator results

$$\pi((Emp(ename, Children(cname))COMPANY)$$

The project list contains two nested subrelations of Emp and Children, followed by another project list of the desired attributes for that subrelation. In this case, the desired attributes for Emp are the atomic attribute of ename and the subrelation of Children, with the desired attributes for Children being the atomic attribute of cname. This project list precedes the desired relation name, COMPANY, in which the project is to operate. The resultant set of 2 tuples are shown in Figure 3.3. Smith and Jones are members of the first tuple and Adams, Carter, and Davis are members of the second tuple.

3.1.2.3 Nest (ν). The operators nest and unnest restructure or change the way tuples in a relation are grouped together in a nested relation. The nest operator and the unnest operator, described in the next section, restructure only the subrelation that is specified without affecting any of the other attributes in the relation. The nest operator groups targeted attributes situated at the same level of nesting which agree on all the attributes that are not in the targeted attributes, and

COMPANY

dno	dname	eno	sal	ename

Figure 3.4. 1NF Database Table header

nests this group one level deeper within the relation. The nest list identifies the targeted attributes, atomic and relation-valued, which are tc be grouped together for the new subrelation.

The nest operator requires three elements. First, a list of attributes that are to be nested at a deeper level. Second, the name of the relation or subrelation that currently contains the attributes in the list to be nested. The last element is the name of the relation-valued attribute that the attributes will be nested under. The generic format of the nest operator is as follows:

$$\nu(Relation\ (Nest\ list) \rightarrow New\ Subrelation)$$

As an example, to nest the attributes, (eno, ename, sal), of the COMPANY relation of Figure 3.4 into the subrelation Emp to form the corresponding relation COMPANY in Figure 3.5, the following statement would be required:

$$\nu(COMPANY(eno, ename, sal) \rightarrow Emp)$$

The nest list, (eno, ename, sal), contains the names of the attributes that are to be nested. This is preceded by the name of the relation, COMPANY, which currently contains the attributes in the nest list. The nest list is then followed by the name of the new subrelation, Emp, that will now contain the attributes in the nest list.

3.1.2.4 Unnest (μ). The unnest operator performs the reverse of the nest operator. This is done by 'ungrouping' or flattening out the subrelation identified in the unnest list into the

COMPANY

dno	dname	Emp		
		eno	sal	ename

Figure 3.5. *NEST* operator results

level above it. Multiple instances of the outer level are generated for each tuple that was originally associated with it in the nested form. This operator only requires two elements. First, the name of the relation that contains the subrelation to be unnested. Second, an unnest list which can contain the single name of a top-level subrelation or another unnest list which contains a lower level subrelation that is to be flattened back to the 1NF. The generic format of the unnest operator is as follows:

$$\mu(Relation(Unnest\ list))$$

As an example, to unnest the relation found in Figure 3.5 back to the original relation found in Figure 3.4 the following statement is used:

$$\mu(COMPANY(Emp))$$

The unnest list, (EMP), contains the nested subrelation to be flattened. This is preceded by the relation name, COMPANY, which will now contain the flattened version of the nested subrelation.

3.1.2.5 Cartesian-Product (×). The cartesian product operator allows cross-products to be performed between a relation or a relation-valued attribute and another relation. This allows us to operate on tuples of a relation valued attribute nested deep inside the structure of the

REL3

X	Y	
	Z	A
x1	z1	a1
	z2	a2
x2	z3	a2

Figure 3.6. Nested Database relation

REL4

Z	W	
	S	T
z1	s1	t1
	s2	t1
z3	s2	t2

Figure 3.7. Nested Database relation

relation scheme. The result of this operation is a relation whose scheme has the attributes of both the relations. Note: this operation is defined so that one operand can be a relation or a subrelation while the other operand has to be a relation, since it is illogical to obtain a cross-product of two subrelations.

The cartesian-product operator requires three elements to perform the cross-product of two relations. The first two are the names of the relations or subrelations to be acted upon (see Note above). The third element is the join path which is defined in Appendix B. This path indicates the location of the attributes that the two relations are operated upon, in other words it is the navigator used to reach the nested attributes of the subrelation required for the cross-product. The generic format of the Cartesian-product operator is as follows:

$$\times(Relation1(JoinPath), Relation2)$$

X	Y				
	Z	A	Z'	W	
				S	T
x1	z1	a1	z1	s1	t1
				s2	t1
	z1	a1	z3	s2	t2
	z2	a2	z1	s1	t1
				s2	t1
	z2	a2	z3	s2	t2
x2	z3	a2	z1	s1	t1
				s2	t1
	z3	a2	z3	s2	t2

Figure 3.8. Result from Cartesian-Product

An example of the cartesian-product operator to obtain the cross product of REL3 of Figure 3.6 and REL4 of Figure 3.7 would be formulated as follows:

$$\times(REL3(Y), REL4)$$

This would form the cross product using the Y subrelation of REL3 as the operand along with REL4. See Figure 3.8 for result of operation.

3.1.2.6 Join (⋈). The join operator has essentially the same properties and results as the cartesian-product operator (see section 3.1.2.5) followed by a selection (see section 3.1.2.1) on the tuples that agree on the values of their common attributes. The generic format of the join operator would be as follows:

$$\bowtie (Relation1(Join\ Path), Relation2)$$

An example of the join operator to obtain the join of REL3 of Figure 3.6 and REL1 of Figure 3.9 would be formulated as follows:

REL1

A	B	C			
		D		E	F
		G	H		
a1	b1	g1	h1	e1	f1
		g2	h2		
		g3	h3	e2	f1
		g4	h4		
a2	b1	g5	h1	e3	f2
		g6	h2		
		g6	h4	e4	f3

Figure 3.9. Nested Database Relation

X	Y						
	Z	A	B	C			
				D		E	F
				G	H		
x1	z1	a1	b1	g1	h1	e1	f1
				g2	h2		
				g3	h3	e2	f1
				g4	h4		
	z2	a2	b1	g5	h1	e3	f2
				g6	h2		
				g6	h4	e4	f3
x2	z3	a2	b1	g5	h1	e3	f2
				g6	h2		
				g6	h4	e4	f3

Figure 3.10. Result from Join operator

$$\bowtie (REL3(Y), REL1)$$

This would form the join using the Y subrelation of REL3 as the operand along with REL1. See Figure 3.10 for result of operation.

3.1.2.7 Extended Binary Operators. The binary operators union, difference, and intersection normally operate on **entire tuples** . These operators take two relations which have the

REL2

A	B	C			
		D		E	F
		G	H		
a1	b1	g2	h2	e1	f1
		g3	h4	e2	f1
a3	b2	g4	h2	e2	f2
		g5	h2		
a2	b1	g6	h4	e4	f3

Figure 3.11. Nested Database Relation

same relation scheme and return the union, difference, and intersection of the relations as defined in normal set theory. The scheme of the resultant relation from these operations will be the same as that of the relations involved in the operation. The extended versions of these operators (op^e) allow us to perform the operation between two relations so that the associated tuples of any subrelations are also operated upon. This is done by testing each attribute at every level of one relation and comparing it to each corresponding attribute at every level of the second relation. The relations in Figure 3.9 and Figure 3.11 are used to illustrate these extended operators and the corresponding results are shown in figures 3.12, 3.13, and 3.14.

- Union ($\cup, \cup^e$).

  The union of two relations in "standard" relational algebra produces those tuples which are contained in both of the operand relations. This is also true for the extended version of the operator, including the associated tuples of any subrelation. The operator requires only two elements, both of which are the names of the relations to be operated upon. For example, to perform a union of all the tuples of REL1 and REL2 the following statement is required.

$$\cup^e(REL1, REL2)$$

A	B	C			
		D		E	F
		G	H		
a1	b1	g2	h2	e1	f1
		g1	h1		
		g3	h4	e2	f1
		g3	h3		
		g4	h4		
a3	b2	g4	h2	e2	f2
		g5	h2		
a2	b1	g6	h4	e4	f3
		g5	h1	e3	f2
		g6	h2		

Figure 3.12. Result from Union operation

The result is shown in Figure 3.12. Without the extension the result would contain multiple listings of the two a1 tuples from REL1 and REL2 because the nested values would not be considered.

- Difference ($-, -^e$).

The difference of two relations in "standard" relational algebra produces those tuples which are not common to both of the operand relations. That is all tuples which are in the first relation which are not in the second relation. This is also true for the extended version of the operator, including the associated tuples of any subrelation. The operator requires only two elements, both of which are the names of the relations to be operated upon. For example, to perform the difference of all the tuples of REL1 and REL2 the following statement is required.

$$-^e(REL1, REL2)$$

The result is shown in Figure 3.13. Without the extension the result would be an empty set because the A and B values from REL1 and REL2 are the same except for a3. Therefor

A	B	C			
		D		E	F
		G	H		
a1	b1	g1	h1	e1	f1
		g2	h2		
		g3	h3	e2	f1
		g4	h4		
a2	b1	g5	h1	e3	f2
		g6	h2		

Figure 3.13. Result from Difference operation

A	B	C			
		D		E	F
		G	H		
a2	b1	g6	h4	e4	f3

Figure 3.14. Result from Intersection operator

there are no tuples that are in REL1 but not in REL2 because the nested values would not be considered.

- Intersection ($\cap, \cap^e$).

The intersection of two relations in "standard" relational algebra produces those tuples which are common to both of the operand relations. This is also true for the extended version of the operator, including the associated tuples of any subrelation. The operator requires only two elements, both of which are the names of the relations to be operated upon. For example, to perform an intersection of all the tuples of REL1 and REL2 the following statement is required:

$$\cap^e(REL1, REL2)$$

The result is shown in Figure 3.14. Without the extension the result would contain all listings of the two a1 tuples and a2 tuples from REL1 and REL2 because the nested values would not be considered only the top most attribute values (A and B).

3.2 SQL/NF

3.2.1 Introduction. Roth, Korth and Batory (13) have proposed an extension to SQL called SQL/NF. This extension has all the power of standard SQL as well as the ability to define nested relations in the data definition language, and query these relations directly in the nested form. In addition to adding the nest and unnest operations to the language, they also modified the existing language to handle accessing nested relations. The method used by SQL to define relations was also modified for defining nested relations. In this section we will describe the SQL/NF query language as defined in (13). The BNF for the language can be found in Appendix C.

Before we introduce the SQL/NF language itself, let us introduce two sample nested databases (see Figure 3.15). We will use variations of these data bases as examples to help better understand the language definitions. First, we have an employee relation, EMPLOYEE, with attributes (NAME), (AGE), department number (DNO), (CHILDREN), and (PROJECTS). The CHILDREN relation in each EMPLOYEE tuple has attributes (NAME), (AGE), and (TOYS). The TOYS subrelation in each CHILDREN tuple has attributes (NAME), and (COLOR). The PROJECTS subrelation in each EMPLOYEE tuple has attributes (NAME), and (NUMBER). Next, we have a company relation, COMP, with attributes department number (DNO), department name (DNAME), location (LOC), and employee (EMP). The EMP subrelation in each COMP tuple has attributes employee number (ENO), employee name (ENAME), salary (SAL), and (CHILDREN). The CHILDREN subrelation in each EMP tuple has attributes child name (CNAME), and date of birth (DOB).

3.2.2 SQL/NF Query Facilities. The basic structure of an SQL/NF expression consists of three main clauses: *SELECT, FROM,* and *WHERE.*

- The *SELECT* clause contains the list of attributes, from the relations in the *FROM* clause, desired as a result of the query.

EMPLOYEE

NAME	AGE	DNO	CHILDREN				PROJECTS	
			NAME	AGE	TOYS		NAME	NUMBER
					NAME	COLOR		

COMP

DNO	DNAME	LOC	EMP				
			ENO	ENAME	SAL	CHILDREN	
						CNAME	DOB

Figure 3.15. EMPLOYEE and COMP Relations

- The *FROM* clause contains the list of relations to be used to produce the desired query results.
- The *WHERE* clause contains a list of predicates which qualify the selection of tuples from the relations in the *FROM* clause.

The basic construct of an SQL/NF query is as follows:

SELECT attribute-list

FROM relation-list

WHERE predicate-list;

This select-from-where(SFW) expression is conceptually executed by performing a Cartesian product of all the relations in the relation list, selecting only the tuples that meet the conditions in the predicate list, and then projecting only the attributes in the attribute list. If no tuple selection conditions are required, the *WHERE* clause can be omitted. If all attributes of the relation list are desired then the attribute list can be replaced with the key-word *ALL*, to conform with the proposed standard relational database language known as RDL (1)

SQL/NF also allows an even easier way to access the entire contents of a relation by replacing the "*SELECT ALL FROM* relation-name" clause with only the "relation-name." For example, to select all the tuples from the COMP relation in department number 123 the following query would be used:

```
SELECT ALL
FROM   COMP
WHERE DNO = 123;
```

Or using the simplified notation:

```
COMP WHERE DNO = 123;
```

Sometimes it is easier for the user to list the attributes which are not desired instead of listing all the attributes desired. For this, SQL/NF allows the construct "*ALL BUT* attribute-list". As an example, if the user desires to obtain all tuples in the EMPLOYEE relation which have a department number 123 it would be redundant to list the DNO attribute for all the tuples therefore the following query would list all the attributes but DNO:

```
SELECT ALL BUT DNO
FROM   EMPLOYEE
WHERE DNO = 123;
```

3.2.2.1 Nested Query Expressions. The principle of orthogonality has been usefully employed in defining the nested data structure. Wherever a scalar (atomic) valued attribute could occur in a 1NF relation, a relation valued attribute can now occur. SQL/NF has the closure property where the result of any query on one or more relations can also be considered a relation. The principle of orthogonality allows a SFW-expression wherever a relation name could be used. Therefore, in SQL/NF SFW-expressions are allowed in the *FROM* clause. For example, consider the following query:

```
SELECT NAME
FROM   COMP, (SELECT ALL
              FROM   EMPLOYEE
              WHERE AGE > 35)
WHERE COMP.DNO = EMPLOYEE.DNO AND DNAME = "SUPPLY";
```

This query would produce the name of all the employees in the SUPPLY dept who are over 35 years of age. The SFW-expression in the *FROM* clause produces a relation with the same attributes as the EMPLOYEE relation but contain only the tuples that have a value > 35 for the AGE attribute. Then a Cartesian product is performed with the resultant relation and the COMP relation. The tuples which meet the conditions of the *WHERE* clause are then selected with the NAME attribute from the EMPLOYEE relation being projected out. An equivalent SQL/NF query to the above query which is more appropriate to what is actually taking place in the *FROM* clause would be as follows:

```
SELECT NAME
FROM   COMP, (EMPLOYEE WHERE AGE > 35)
WHERE COMP.DNO = EMPLOYEE.DNO AND DNAME = "SUPPLY";
```

A more complex example containing nested query expressions in the SQL *FROM* clause involves the *UNION* operator. The standard SQL/NF format for using the *UNION* operator is as follows:

```
query-expression UNION query-expression;
```

For example, given two relations COMP_A and COMP_B which have the same scheme as the COMP relation. If these two companies merge and it is desired to merge the two relations the following SQL/NF query would be used:

```
COMP_A UNION COMP_B;
```

If a condition is set on what tuples are desired in performing the *UNION*, then in SQL the same condition must be in the *WHERE* clause for both SFW-expressions. For example, given the previous query with the condition that only the tuples with a location of "DAYTON" are desired, then the SQL/NF query would be:

```
SELECT ALL
FROM   (COMP_A UNION COMP_B)
WHERE  LOC = "DAYTON";
```

SFW-expressions in the *SELECT* clause are applicable when using a nested database. This is because attributes in a nested database can be relation valued attributes, therefore, subrelation names can occur in the *SELECT* clause and under the principle of orthogonality SFW-expressions can also occur. The only restriction is that the relation name used in the *SELECT* clause must be a relation valued attribute of one of the relations identified in the *FROM* clause. For example, if all the information contained in the relation valued attribute PROJECTS of the EMPLOYEE relation along with the employees name, the following query could be used:

```
SELECT NAME, PROJECTS
FROM   EMPLOYEE;
```

This query shows how a subrelation name (PROJECTS) would appear in the *SELECT* clause.

If the user requires to know which employees are working on project "X" then a condition must be set on project name and the following query could be used:

```
SELECT NAME, (SELECT NUMBER
              FROM   PROJECTS
              WHERE  NAME = "X")
FROM   EMPLOYEE;
```

This query shows how a SFW-expression would appear in the *SELECT* clause. Note that PROJECTS is a relation valued attribute of EMPLOYEE and that the nested *WHERE* clause condition only applies to the "NAME" attribute of the subrelation PROJECTS.

To illustrate a more complex nested SFW-expression in the *SELECT* clause, let's query the EMPLOYEE relation for the name of the toys that employee Smith's children have:

```
SELECT (SELECT (SELECT NAME
                FROM   TOYS)
        FROM   CHILDREN)
FROM   EMPLOYEE
WHERE  NAME = "SMITH";
```

Note that SF-expressions in the *SELECT* clause are enclosed by "( )" for each level of nesting and can only list attributes from the relation valued attribute identified in the associated *FROM* clause. Since the NAME attribute under the TOYS subrelation is at the third level of nesting, three nested *SELECT* statements are required to reach this level.

Nested SFW-expressions in the SQL/NF *WHERE* clause follow the same requirements as in SQL. That is, the SFW-expression can't stand alone in the *WHERE* clause, it must be an element of:

1. Comparison predicate.
2. BETWEEN predicate.
3. IN predicate.
4. LIKE predicate.
5. EXISTS predicate.
6. NULL predicate.

Consider the following example using the EXISTS predicate:

```
SELECT DNAME, EMP
FROM   COMP
WHERE  EXISTS (EMP WHERE SAL < 25000);
```

The result of the above query on the COMP relation would produce all the department names along with all the information contained in the EMP subrelation for all the departments that have at least one EMP salary value less than 25000.

3.2.2.2 Functions. In SQL, the argument to a function like AVG (average) can only be an atomic valued attribute column in a relation and the result is a single scalar value. SQL/NF has expanded the argument for functions to include relation valued attributes. Then, by the principle of orthogonality, the argument can include any expression that evaluates to a relation.

Consider the SQL query to obtain the average age of the employees in the EMPLOYEE relation:

```
SELECT AVG(AGE)
FROM   EMPLOYEE;
```

The argument to the *AVG* function is the entire AGE column of the EMPLOYEE relation, and there is no vehicle to place a condition on selection of tuples averaged by the function (e.g. *WHERE* AGE > 25). This is remedied in SQL/NF by applying the desired relation to the function and not just a single attribute. Therefore the SQL/NF equivalent to the above query would be:

```
AVG (SELECT AGE
      FROM   EMPLOYEE);
```

The same query with the condition that the average age be determined for those employees over 25 would be:

```
AVG (SELECT AGE
      FROM   EMPLOYEE
      WHERE  AGE > 25);
```

Now let us consider a more complicated query where we want to determine the total amount made by all the employees in the COMP relation, who are in the Engineering department and make less than $50000:

```
SELECT (SUM (SELECT SAL
              FROM   EMP
              WHERE  SAL < 50000)
FROM   COMP
WHERE  DNAME = "ENGINEERING");
```

Another example, utilizing a function in the *WHERE* clause, is the query which identifies all the departments in the COMP relation that have more than 5 employees:

```
SELECT DNO
FROM   COMP
WHERE  COUNT(EMP) > 5;
```

In SQL, queries of this type are usually formulated using *GROUP BY* and *HAVING* clauses. These are some of the hardest to formulate in SQL, and operate under a different set of rules from standard SQL queries. Due to the ability of SQL/NF to allow nested queries in the *SELECT* clause and the structuring ability of the nested model to already have attributes "grouped by", *GROUP BY* and *HAVING* are totally unnecessary. By structuring relations appropriately, we can turn any previous *GROUP BY* or *HAVING* query into a straight-forward SFW-expression. The elimination of *GROUP BY* and *HAVING* is a major advantage of the nested model(13).

An additional advantage of SQL/NF's ability to accept relations or nested SFW-expressions as input to functions is to apply the function to several attributes simultaneously to a multi-attribute relation. The following example of this ability was given in (13): Suppose we have a Sales relation with employee number(eno) and 12 sales attributes (Jan-sales, Feb-sales, ..., Dec-sales) showing total sales for each month of the year for the employee. Then to get the total of all sales in each month we can write:

```
SUM (SELECT Jan-sales, Feb-sales, Mar-sales, Apr-sales,
            May-sales, Jun-sales, Jul-sales, Aug-sales,
            Sep-sales, Oct-sales, Nov-sales, Dec-sales
       FROM Sales);
```

Or using "*ALL BUT* attribute-list" the query can be simplified to:

```
SUM (SELECT ALL BUT eno
       FROM Sales);
```

The *SUM* function is applied to each column of the argument relation. In general a column function, (*SUM, AVG, MAX, MIN*), reduces a relation to a single tuple with the same number of attributes, by applying the function to each column of the relation. A table function(*COUNT*), reduces a relation to a single tuple with one attribute. Thus, the result of applying a function is always a single tuple relation (13).

When a column or table function is applied to a nested relation it does not leave the result in a nested form. Because it would not make sense to retain the relation structure for a single tuple.

Therefore the result from a function applied to a nested relation is unnested one level.

3.2.2.3 NULL values and operations dealing with NULLs. The treatment of NULL values encountered in a tuple by functions in SQL/NF is different from SQL. In SQL/NF an error condition is raised when a NULL value is encountered while in SQL the value is simply ignored. This forces the SQL/NF user to remove any NULLs before applying the function and prevents inaccurate results from a query over a database that contains NULL values.

SQL/NF provides a method for dealing with subsumed tuples which is not available in SQL. A subsumed tuple is like a duplicate tuple as they do not provide any more information than some other tunle in the relation. For example the tuple t=<SMITH, NULL, NULL> is "subsumed" by the tuple s=<SMITH, 123, MKT> and the tuple r=<SMITH, NULL, ENG> while it is not "subsumed" by the tuple q=<Jones, 111, ACCT>. The information in tuples *s* and *r* contain all the information in tuple *t* therefore it can be "subsumed" and considered a duplicate tuple. The elimination of duplicate tuples in SQL/NF is accomplished the same way as SQL by using the function *DISTINCT*, but this function does not handle subsumed tuples. The SQL/NF function *SUBSUME* can be used to eliminate tuples that are "subsumed". For example, to get employee names and project names from the EMPLOYEE relation and eliminate any subsumed PROJECTS tuples, we would use the following query:

```
SELECT NAME, SUBSUME (SELECT NAME
                      FROM   PROJECTS)
FROM   EMPLOYEE;
```

Another method available in SQL/NF for dealing with NULL values is the *PRESERVE* clause. This clause is useful when performing a "join" on two or more relations, identified in the *FROM* clause, based on conditions in the *WHERE* clause and it is desired to "preserve" the tuples of one of the relations that do not meet the conditions of the *WHERE* clause. The tuples of the relation in the *PRESERVE* clause are included with the results of the SFW-expression with the attribute values of the other relations set to NULL for the "preserved" tuples. For example, if the two relations EMPLOYEE and COMP are joined on the DNO common attribute and it is desired to also include any tuple in the COMP relation that does not have any matching EMPLOYEE tuples we would use the following query:

```
SELECT    NAME, AGE, DNO, DNAME, LOC
FROM      COMP, EMPLOYEE
WHERE     COMP.DNO = EMPLOYEE.DNO
PRESERVE COMP;
```

The result of this query would produce a relation with the attributes NAME, AGE, DNO, DNAME, LOC. Matching the tuples in the COMP relation and EMPLOYEE relation with the same DNO value. In addition, any tuple in the COMP relation that does not have a corresponding DNO value in the EMPLOYEE relation will be included in the result with the attribute values for NAME and AGE set to NULL.

3.2.2.4 Don't Care values. When comparing tuple literals with attribute values in the *WHERE* clause, SQL/NF has provided for a "wild card" value in the tuple literal. The wild card value is the question mark (?). The "don't care" question mark can be used for any constant value in a tuple literal. For example, to find the name of all the employees in the EMPLOYEE relation who have worked on project "X" , we could use the following query:

```
SELECT NAME
FROM   EMPLOYEE
WHERE <"X", ?> IN PROJECTS;
```

3.2.2.5 Data and relation restructuring operations. The restructuring operations provided in SQL/NF for nested relations correspond directly to the *NEST* and *UNNEST* algebra operators as identified in section 2.2.3. The syntax for these operators is as follows:

```
NEST table-name
ON    attribute-list [AS column-name];

UNNEST table-name
ON       attribute-list;
```

Where "table-name" can be replaced, via the principle of orthogonality, with "(query-expression)". The *NEST* operator provides for giving the newly created nested relation a name (column-name). However, if the name is left out the nested relation can't be referenced elsewhere in the query.

Let us consider restructuring the COMPANY relation in Figure 3.4 by nesting the attributes (eno, sal, ename) in a relation valued attribute Emp. This will be accomplished by the following query:

```
NEST COMPANY
ON   eno, sal, ename AS Emp;
```

The resultant structure from this query can be seen in Figure 3.5. To restructure the COMPANY relation in Figure 3.5 to the COMPANY relation in Figure 3.4 the following query would be used:

```
UNNEST COMPANY
ON     Emp;
```

Consider the three 1NF database relation tables in Figure 3.16, these tables can be combined into the single nested relation COMP (Figure 3.15) by nesting the CHILD relation within the Employee relation and nesting that result within the DEPT relation based on the common attributes DNO and ENO. This can be accomplished by the following query:

```
SELECT ALL BUT Employee.DNO
FROM   DEPT, (NEST (SELECT ALL BUT CHILD.ENO
                    FROM   EMP, (NEST CHILD
                                 ON   Cname, DOB AS CHILDREN)
                    WHERE Employee.ENO = CHILD.ENO)
              ON   ENO, ENAME, SAL, CHILDREN AS EMP)
WHERE  DEPT.DNO = Employee.DNO;
```

To obtain the three relation tables in Figure 3.16 from the COMP relation we would use the queries.

```
SELECT (SELECT ENO, CNAME  DOB
        FROM   (UNNEST EMP
                ON      CHILDREN))
FROM   COMP;
```

DEPT

DNO	DNAME	LOC

Employee

ENO	ENAME	DNO	SAL

CHILD

ENO	CNAME	DOB

Figure 3.16. Three Sample 1NF Relations

```
SELECT ENO, ENAME, DNO, SAL
FROM   (UNNEST COMP
         ON       EMP);

SELECT DNO, DNAME, LOC
FROM   COMP;
```

The first query would produce the CHILD relation from COMP, the second query would produce the Employee relation from COMP, and the last query would produce the COMPANY relation from COMP.

One other restructuring operation known in SQL as the *ORDER BY* clause is also available in SQL/NF, with a change in syntax. The function rearranges the tuples of a relation in ascending (*ASC*) or descending (*DESC*) order based on identified attributes (column-name). The syntax change follows the same structure as the other functions in SQL/NF as follows:

```
ORDER table-name
BY      column-name [ASC|DESC] {, column-name [ASC|DESC]}
```

Where "table-name" can be replaced, via the principle of orthogonality, with "(query-expression)". For example, to obtain a list of tuples from the COMP relation which are in ascending order based on the department name would be as follows:

```
ORDER COMP
BY    NAME ASC;
```

3.2.2.6 Name inheritance and aliasing. Sometimes a problem occurs when attribute names of relations used in the *FROM* clause of a SFW-expression are not unique. This can be a result of one or more of the following:

1. The need for multiple copies of the same relation being used in a SFW-expression.
2. Attributes with the same name in two or more of the relations in a SFW-expression.
3. The same name is used in nested relations within a relation.

The solution to the first condition is provided by SQL/NF by using a user defined reference name instead of the original table name. This reference name is identified by using the key word *AS* followed by the reference name. The syntax being as follows:

table-name *AS* reference-name;

Where "table-name" can be replaced, via the principle of orthogonality, with "(query-expression)". For example, to obtain pairs of all the departments in the COMP relation that are at the same location the following query could be used:

```
SELECT COMP_A.DNAME, COMP_B.DNAME
FROM   COMP AS COMP_A, COMP AS COMP_B
WHERE  COMP_A.LOC = COMP_B.LOC
       AND COMP_A.DNO < COMP_B.DNO;
```

The solution SQL/NF provides for the second condition is similar to the previous solution in that a reference name is also used. However, this reference name is provided by the original table name and not the user. For example, let the DNAME attribute in the COMP relation be called NAME. If both the COMP and EMPLOYEE relations are identified in the *FROM* clause of a SFW-expression and the NAME attribute from COMP and the NAME attribute from EMPLOYEE is desired in the *SELECT* clause, they can be uniquely identified as COMP.NAME and EMPLOYEE.NAME.

The third condition usually appears when the *UNNEST* operator is used. For example, in the EMPLOYEE relation NAME is used at the top level as well as in both nested relations of CHILDREN and PROJECTS. If the PROJECTS nested relation is unnested the result would produce two attributes with the same name (NAME). SQL/NF avoids this by identifying the unnested NAME attribute as PROJECTS.NAME using a reference name from the original relation name (PROJECTS). Reference names can also be used to simplify query statements by naming a nested query statement and using that name elsewhere with the SFW-expression. For example, remember the example query for the *EXISTS* predicate in section 3.2.2.1 where the nested query expression (EMP *WHERE* SAL < 2500) was used in both the *SELECT* clause and the *WHERE* clause. This query can be simplified using a reference name (SAL_limit) as follows:

```
SELECT (EMP WHERE SAL < 25000) AS SAL_limit
FROM   COMP
WHERE  EXISTS (SAL_limit);
```

3.2.3 Data Definition Language(DDL). The defining of relations in SQL is accomplished via the *CREATE TABLE* command, and the defining of view is accomplished via the *CREATE VIEW* command. The *CREATE TABLE* command incorporates a list of attribute names and their respective domains (e.g. CHAR, INT, etc.) to define the structure of a relation. Our version of SQL/NF [1], as defined in the BNF of Appendix C, uses a modified version of this *CREATE TABLE* command to allow for nested relations. The SQL/NF version also incorporates a variety of integrity constraints as proposed and defined in the RDL standard(1). These constraints can be specified along with the attribute names and domains and include (UNIQUE, NOT NULL, REFERENCES, CHECK, ...).

Let us continue our explanation of the *CREATE TABLE* command by providing an example for creating the DEPT relation in Figure 3.16:

[1]Note: Our version is different from the Data Definition Language defined in (13).

```
CREATE TABLE DEPT
                  (DNO INT 1 UNIQUE NOT NULL,
                   DNAME CHAR 32,
                   LOC CHAR 24);
```

The key words *CREATE TABLE* are followed by the name of the relation (DEPT) being defined. This is followed by a list of items which contains the attribute names (DNO, DNAME, LOC) and their associated domains. The DNO attribute also has the constraints *UNIQUE NOT NULL* in the column definition which identify it as the "key" to the relation, for all the values must be unique and can't be a NULL value. When a nested relation is desired to be created, the attribute item is replaced by a nested TABLE statement which consists of the key word *TABLE* followed by the nested table name and the list of attribute items as in the *CREATE TABLE* command. For example, to create the nested relation COMP in Figure 3.15 the following *CREATE TABLE* command would be used:

```
CREATE TABLE COMP
                  (DNO INT 1 UNIQUE NOT NULL,
                   DNAME CHAR 32,
                   LOC CHAR 24,
                   (TABLE EMP
                          (ENO INT 1 UNIQUE,
                           ENAME CHAR 32,
                           SAL FLOAT 4,
                           (TABLE CHILDREN
                                  (CNAME CHAR 24,
                                   DOB CHAR 7)))));
```

In order to simplify the definition of nested relations, SQL/NF allows the definition of relation "schemes" separately from the definition of the relations themselves. This is accomplished by the *CREATE TYPE* command, which follows the same format as the *CREATE TABLE* command. The only difference between these two commands is that the *CREATE TYPE* command specifies table definitions without actually creating a table. The scheme name can then be used in place of the list of attribute items, thus simplifying the definition. This command is very useful when deeply nested relations are being defined or when the same type of nested relation appears more than once in a relation definition. For example, to create the COMP relation a scheme can be defined for the CHILDREN relation and the EMP relation and then used in defining the COMP relation as follows:

```
CREATE TYPE CHILD
                (CNAME CHAR 24,
                 DOB CHAR 7);

CREATE TYPE EMPLOYEE
                (ENO INT 1 UNIQUE,
                 ENAME CHAR 32,
                 SAL FLOAT 4,
                 (TABLE CHILDREN CHILD));

CREATE TABLE COMP
                (DNO INT 1 UNIQUE NOT NULL,
                 DNAME CHAR 32,
                 LOC CHAR 24,
                 (TABLE EMP EMPLOYEE));
```

The SQL *CREATE VIEW* command is also provided in SQL/NF to provide views of nested relations. The syntax is as follows:

CREATE VIEW table-name *AS* query-expression

For example, to create a view of the COMP relation which does not include the SAL attribute in the EMP relation valued attribute would be defined as follows:

```
CREATE VIEW Comp_WO_SAL AS
                SELECT DNO, DNAME, LOC, (SELECT ALL BUT SAL
                                         FROM EMP)
                FROM    COMP;
```

The *DROP TABLE, DROP TYPE, DROP VIEW* commands all operate in the same way, in that they simply delete a Table, Type, or View from the symbol table defined by a DDL statement. The commands all follow the same syntax rules which is simply to identify the name of what is to be deleted following the key words. For example, to delete the relation table COMP from the symbol table the following command would be used:

DROP TABLE COMP;

3.2.4 Data Manipulation Language (DML). In this section we discuss the commands used in SQL/NF to store, modify, and erase data from relations in a database. The DML commands can be thought of as functions, for they produce relations from relations with the added effect of replacing the old relation with the resultant relation. The syntax for these commands (*STORE, MODIFY, ERASE*) is adapted from the RDL standard (1) with the additional ability to work with nested relations.

The *STORE* command is used to add new tuples to a relation via a user defined set of tuples or via a query specification to another relation. For example, to add a two new employees to the Employee relation in Figure 3.16 the command would be:

```
STORE Employee
        VALUES <123, SMITH, 111, 25000>
                <124, JONES, 222, 18000>;
```

The command also has an option where the user can specify which attributes are to be used. This is accomplished by an attribute-list following the relation name. Consider the newly created CHILD relation in Figure 3.16. If the user wants to enter all the ENO values from the Employee relation into the CHILD relation without entering them by hand, the following command statement can be used:

```
STORE CHILD (ENO)
        SELECT ENO
        FROM   Employee;
```

As each new tuple is stored in the CHILD relation, the value for ENO will be obtained from the Employee relation and the values for CNAME and DOB will be set to the default value defined in the *CREATE TABLE* definition or to NULL if no default is specified.

When dealing with a nested relation, the attribute-list will reflect the location of the attributes desired by identifying the name of the relation valued attribute followed by an attribute-list for that relation[2]. For example, to store a new department in the COMP relation the following command

[2]Note: Our version of the Data Manipulation Language is different from the DML defined in (13).

COMP

DNO	DNAME	LOC	EMP				
			ENO	ENAME	SAL	CHILDREN	
						CNAME	DOB
001	ENG	BLDG3	111	SMITH	12000	JANE	MAY
						DAVE	APRIL
			121	JONES	12500	BOB	OCT
						SUE	JAN

Figure 3.17. Result from *STORE* command

could be used:

```
STORE COMP (DNO, DNAME, LOC, EMP(ENO, ENAME, SAL, CHILDREN(CNAME, DOB)))
      VALUES <001, ENG, BLDG3, (<111, SMITH, 12000, (<JANE, MAY>
                                                     <DAVE, APRIL>) >
                                <121, JONES, 12500, (<BOB, OCT>
                                                     <SUE, JAN>) >) >;
```

The result from adding the new tuple to the COMP relation can be seen in Figure 3.17. Notice that within the single COMP tuple there are 2 tuples stored in the EMP nested relation an 2 tuples stored in the CHILDREN nested relation for each EMP tuple.

The *MODIFY* command is used to change tuples that already exist in a database. This is accomplished by setting a new value for the tuples and qualifying the change with an optional *WHERE* clause. For example, to insert a child name for employee number 121 in the CHILD relation, we stored earlier with employee numbers, we could use the following command.

```
MODIFY CHILD
       SET CNAME = "SUE"
       WHERE ENO = 121;
```

If we wish to give all the employees in the Employee relation a 10% raise, the *WHERE* clause can be left out and the following command could be used:

```
MODIFY Employee
       SET SAL = SAL*1.1
```

When dealing with nested relations the syntax becomes a little more complicated for the path to the sub-relation being modified needs to be identified as well as the modification. This is done by nesting DML commands in the *SET* clause of the *MODIFY* command. For example, to add a new child to employee Smith in the COMP database of Figure 3.17 the following command could be used:

```
MODIFY COMP
        SET  EMP = (MODIFY EMP
                            SET CHILDREN =(STORE  CHILDREN
                                                    VALUES <BILL, AUG>)
                            WHERE ENO = 111)
        WHERE DNO = 001;
```

The *ERASE* command is used to erase tuples from a relation, qualifying the target tuple by an optional *WHERE* clause. For example, to erase the tuple we modified earlier in the CHILD relation the following command could be used:

```
ERASE CHILD
WHERE ENO = 121;
```

Consider the relations in Figure 3.16, if the user wishes to remove all tuples in the Employee relation which don't have a child in the CHILD relation, the following command could be used:

```
ERASE Employee
WHERE NOT EXISTS (CHILD  WHERE Employee.ENO = CHILD.ENO);
```

If a tuple that is in a nested relation is to be erased the *MODIFY* command must be used in conjunction with the *ERASE* command. For example, if the user wishes to erase the tuple in the CHILDREN nested relation of COMP we added earlier the following command could be used.

```
MODIFY COMP
        SET  EMP = (MODIFY EMP
                            SET CHILDREN = (ERASE  CHILDREN
                                                WHERE  CNAME = "BILL")
                            WHERE ENO = 111)
        WHERE DNO = 001;
```

IV. Design & Implementation

4.1 Introduction

In this chapter we discuss the design process for the SQL/NF translator and the methodology used to implement it. The basic process for translating an SQL/NF query statement to a Colby algebra tree is as shown in Figure 4.1. In addition we discuss the design and implementation process for the symbol table which is part of the EXODUS catalog manager. First, we describe the Lexical Analyzer and Parser process used for both query translation and symbol table creation. Next, we describe the Data Dictionary design and implementation, followed by a description of the SQL/NF query tree generating process, and last we discuss the generation of the Colby algebra tree from the SQL/NF query tree. We will also present some sample SQL/NF queries with their Colby algebra equivalents.

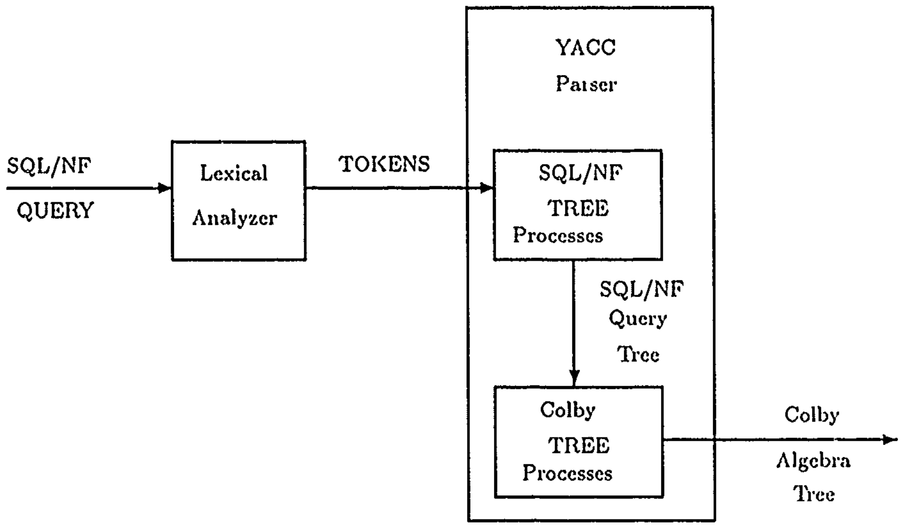

Figure 4.1. Query Tree Process

4.2 Parser

In Section 2.5.1 we identified the UNIX tools of YACC and LEX as our method for implementing the parser component of the EXODUS architechture. The parser process is accomplished

```
⟨ddl statement⟩ ::= ⟨schema⟩ | ⟨scheme⟩
⟨schema⟩ ::= CREATE {⟨table definition⟩ | ⟨view definition⟩}
```

Figure 4.2. SQL/NF sample BNF

in two steps. In the first step, the query is scanned by the lexical analyzer (LEX) which assigns a "token" to the key words and other components of the query statement and returns a stream of these tokens. The second step is accomplished by YACC, which receives the stream of tokens and organizes them according to the input structure rules, when one of these rules is recognized the C code in the *action* part of the rule is invoked.

4.2.1 The LEX process. To implement the LEX process was a straight forward task of creating the file **scanner.l** which contains all the definitions for the key words and other components (punctuation, integers, identifiers, etc.) in the LEX format described in Appendix A. The key words are taken from the SQL/NF BNF found in Appendix C. The program *yylex* is generated using the LEX compiler with the file **scanner.l**.

4.2.2 The YACC process. Implementation of the YACC process was accomplished by first creating the file **parser.y** which contains the translation of the SQL/NF BNF, found in Appendix C, into the YACC format described in Appendix A. This was also a relatively straight forward process, though time consumming, for the BNF format is almost identicle to the YACC format. For example, the BNF in Figure 4.2 is translated into the YACC format shown in Figure 4.3. The key word tokens returned from *yylex* are identified as "T_key-word". The C code that is executed, when a grammar rule is recognized, is enclosed by { and }. The program *yyparse* is generated using the YACC compiler with the file **parser.y**.

In order to verify that the parser was correctly identifying the SQL/NF grammar rules (syntax), only print statements were enclosed in the executable section of each rule (see Figure 4.3). These print statements identified what syntax rules were satisfied for the input query. When testing

```
ddl_statement
            :  schema
               {
                  printf("\nFound DDL STATEMENT");
               }
            |  scheme
               {
                  printf("\nFound DDL STATEMENT");
               }
            ;

schema
            :  T_CREATE table_definition
               {
                 printf("\nFound CREATE TABLE definition");
               }
            |  T_CREATE view_definition
               {
                  printf("\nFound CREATE VIEW definition");
               }
            ;
```

Figure 4.3. YACC format for SQL/NF sample BNF

this version of the parser, conflicts arose from ambiguities in the definitions in the original BNF in (13). In particular, some of the definitions dealing with nested query expressions and table names created problems. Therefore the BNF was modified to accommodate YACC and eliminate any ambiguities. Once this phase was complete we began adding the C code, to the executable section of the rules, to implement the symbol table, Sql query tree, and Colby algebra tree processes.

4.2.3 Parser Input. The first version of the parser was created with the *main* procedure calling *yyparse* which read input directly from standard input and required the user to input all key words in upper case letters. This also limited the structure of the input query statement to a single continuous line of text entered from the keyboard. To correct the first problem, the *main* procedure was modified so that it would convert the input query to all capital letters and store the new version in the file query_file. Then the *main* procedure defines standard input to be the file query_file before calling *yyparse* . However, the modified *main* procedure would not work when

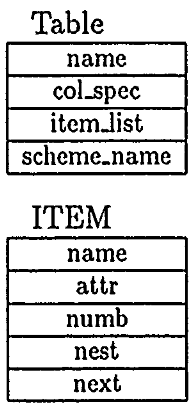

Figure 4.4. The Table and ITEM data structures

standard input was from the keyboard. This problem along with the above second problem was corrected by creating a separate procedure *SQLNF* which reads formatted lines of text entered from the keyboard until a ' ; ' is encountered and stores it in the file **input_file**. Then the procedure calls the *main* procedure redirecting standard input to be from the file **input_file**.

4.3 Catalog Manager

In Section 2.5.2 we identified the composition of the Data Dictionary used by the Catalog Manager to be identical to the one developed by Mankus(10). Therefore, our main task here was to develop the interim data structures and associated processes for the parser to execute for the *CREATE TABLE, CREATE TYPE* , and *CREATE VIEW* DDL statements. The interim data structures are used to hold the information provided in the DDL statements until it can be entered into the Data Dictionary.

The basic construct of a relation table definition consists of the table-name and a list of attribute items or a table-name and a scheme-name. Therefore, the interim data structures we used are called "Table" and "ITEM" and are shown in Figure 4.4. The Table structure holds the name of the table being defined, an identifier as to whether the table consists of an ITEM_LIST or a SCHEME name, a pointer to an ITEM structure which is the beginning of the item-list if the

```
CREATE TABLE COMP
                (DNO INT 1,
                 DNAME CHAR 32,
                 LOC CHAR 24,
                 (TABLE EMP
                         (ENO INT 1,
                          ENAME CHAR 32,
                          SAL FLOAT 4,
                          (TABLE CHILDREN
                                   (CNAME CHAR 24,
                                    DOB CHAR 7)))));
```

Figure 4.5. Sample *CREATE TABLE* command

table consists of one, and the name of the scheme if the table is defined by a scheme name. The ITEM structure holds the attribute name, the domain of the attribute (TABLE, CHAR, INT, or FLOAT), the size of the attribute, a pointer to a Table structure when the attribute is a relation valued attribute, and a pointer to the next ITEM. For example, consider the *CREATE TABLE* command in Figure 4.5 used to create the relation COMP from Figure 3.15. The resultant Table-ITEM tree structure generated by parsing this command is shown in Figure 4.6.

The Data Dictionary developed by Mankus(10) consists of two persistant collections which are tables, a table of relations and a symbol table. The table of relations is a list of the table names, defined by the *CREATE TABLE* command, and an index value to identify the type (scheme) of the relation (see Figure 4.7). The symbol table contains the information on the schemes and their associated attributes, which can be defined by either the *CREATE TABLE* or *CREATE TYPE* commands. When the *CREATE TYPE* command is used there is no entry in the table of relations. When the *CREATE TABLE* command is used in conjunction with a non-scheme name definition, a scheme is created with the same name as the table name. An important feature we included in our Data Dictionary process, is when a table or scheme is defined the name must be unique or it will be rejected.

Some additional commands have been added to the grammar rules in parser.y in order to provide a formatted output of the table of relations and the symbol table. The first command is "DUMP", this simply performs a formatted dump of all the information in the relation and symbol tables similar to Figure 4.7 and Figure 4.8. The second command is "CHECK TABLE table-name", this performs a formatted dump of the symbol table only for the table identified by table-name. If the table-name is left out a dump of the relation table is performed (Figure 4.7). The symbol table dump in Figure 4.8 was created by the command "CHECK TABLE COMP". The last command is "CHECK TYPE scheme-name", this performs a formatted dump of the symbol table only for the scheme identified by scheme-name.

The *CREATE TABLE* tree in Figure 4.6 is used to generate an entry in the Table of Relations and the Symbol Table entries as seen in Figure 4.7 and Figure 4.8 respectively. The Rel Type Index in Figure 4.7 identifies the relation COMP to have a scheme COMP defined by INDEX 42 in Figure 4.8. The relation valued attribute EMP is identified to have a scheme EMPs with INDEX value 52 by the NEST INDEX, the parent relation COMP identified by the PARENT index value 42, and 4 attributes identified by the NUMB value. The SCHEME EMPs was automatically created on-the-fly to provide a scheme type name for the EMP relation valued attribute, which was defined on-the-fly by the *CREATE TABLE* command in Figure 4.5.

An example of a table definition using schemes can be seen in Figure 4.9 for the EMPLOYEE relation found in Figure 3.15. The relation was created using the following set of commands.

```
CREATE TYPE TOY
            (NAME CHAR 32,
             COLOR CHAR 32);

CREATE TYPE CHILD
            (NAME CHAR 32,
             AGE INT 1)
             TABLE TOYS TOY;

CREATE TYPE PROJECT
            (NAME CHAR 32,
             NUMBER INT 2);
```

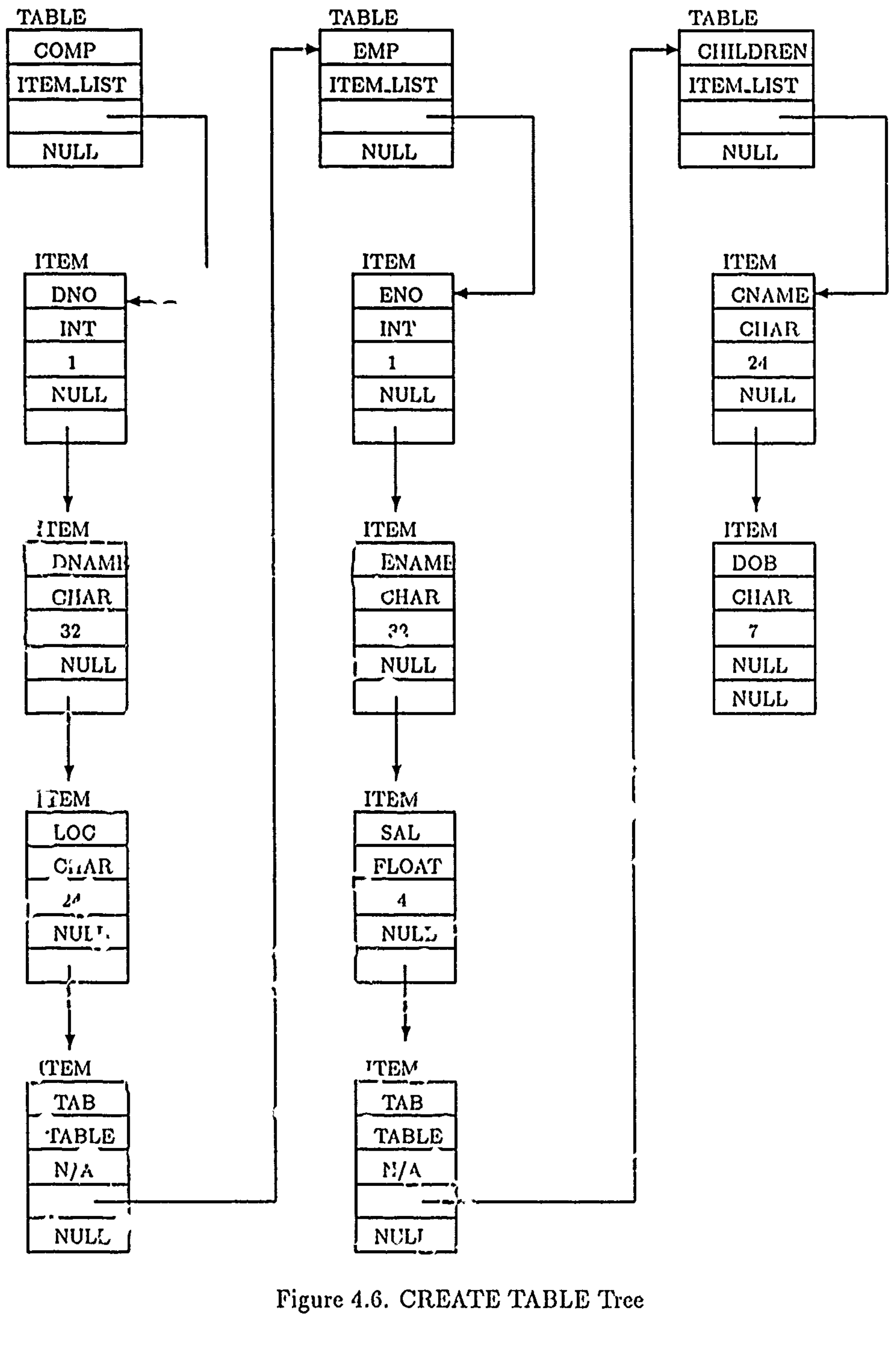

Figure 4.6. CREATE TABLE Tree

```
             Table Of Relations
             ******************

     Rel Index    Rel Name       Rel Type Index
     ---------    --------       --------------
     0            EMP            10
     1            DEPT           16
     2            COMPANIES      20
     3            VIEWD          44
     4            COMP           42
     5            EMPLOYEE       10
```

Figure 4.7. Sample Relation Table

```
                 Dump of Symbol Table for TABLE: COMP
                 ******************************

INDEX  NAME                 LEVEL     DOMAIN          NUMB  PARENT  NEST INDEX
-----  ----                 -----     ------          ----  ------  ----------
42     COMP                 SCHEME    ON_THE_FLY         4      -1          -2
50     DNAME                ATTR      CHAR              32      42          -2
43     DNO                  ATTR      INT                1      42          -2
51     LOC                  ATTR      CHAR              24      42          -2
79     EMP                  ATTR      PREV_DEFINED       4      42          52
52     EMPs                 SCHEME    ON_THE_FLY         4      -1          -2
54     ENO                  ATTR      INT                1      52          -2
55     ENAME                ATTR      CHAR              32      52          -2
56     SAL                  ATTR      FLOAT              4      52          -2
78     CHILDREN             ATTR      PREV_DEFINED       2      52          75
75     CHILDRENs            SCHEME    ON_THE_FLY         2      -1          -2
76     CNAME                ATTR      CHAR              24      75          -2
77     DOB                  ATTR      CHAR               7      75          -2
```

Figure 4.8. COMP Symbol Table

```
                    Dump of Symbol Table for TABLE: EMPLOYEE
                    ******************************

INDEX  NAME                LEVEL     DOMAIN           NUMB  PARENT  NEST INDEX
-----  ----                -----     ------           ----  ------  ----------
10     EMPLOYEE            SCHEME    ON_THE_FLY          5      -1          -2
11     NAME                ATTR      CHAR               32      10          -2
12     AGE                 ATTR      INT                 1      10          -2
13     DNO                 ATTR      INT                 2      10          -2
14     CHILDREN            ATTR      PREV_DEFINED        3      10           3
3      CHILD               SCHEME    ON_THE_FLY          3      -1          -2
4      NAME                ATTR      CHAR               32       3          -2
5      AGE                 ATTR      INT                 1       3          -2
6      TOYS                ATTR      PREV_DEFINED        2       3           0
0      TOY                 SCHEME    ON_THE_FLY          2      -1          -2
1      NAME                ATTR      CHAR               32       0          -2
2      COLOR               ATTR      CHAR               32       0          -2
15     PROJECTS            ATTR      PREV_DEFINED        2      10           7
7      PROJECT             SCHEME    ON_THE_FLY          2      -1          -2
8      NAME                ATTR      CHAR               32       7          -2
9      NUMBER              ATTR      INT                 2       7          -2
```

Figure 4.9. Sample Symbol Table with nested Schemes

```
CREATE TABLE EMPLOYEE
             (NAME CHAR 32,
              AGE INT 1,
              DNO INT 2,
              (TABLE CHILDREN CHILD),
              (TABLE PROJECTS PROJECT));
```

The CHILDREN, TOYS, and PROJECTS relation valued attributes were previously defined as schemes (CHILD, TOY, PROJECT) identified by the NEST INDEX values (3, 0, 7), and these scheme names were used to define the relation valued attributes in the *CREATE TABLE* command.

The DML statements *DROP TABLE* and *DROP TYPE* are implemented by using the table-name or scheme-name to search the relation table and symbol table and remove the table and scheme along with the attributes (children) of the scheme. If a table exists in the relation table that is of the scheme type to be dropped, an error condition is raised and the scheme is not deleted. This is also true if a relation valued attribute is defined to have the scheme type to be

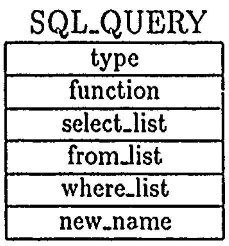

Figure 4.10. The SQL_QUERY node structure

dropped. For example, an attempt to drop the CHILD scheme in the EMPLOYEE symbol table would generate an error condition because it is utilized by the EMPLOYEE scheme.

4.4 SQL/NF Query Tree

The basic construct of an SQL/NF query statement is the three clauses *SELECT, FROM* (which contain elements separated by commas) and *WHERE* (which contain elements separated by *AND* or *OR*). Therefore, the data structures we used for the SQL/NF query tree consist of an SQL_QUERY node, SELECT_NODE, FROM_NODE, and WHERE_NODE to represent each element of their associated clause. These nodes are generated by associated procedures that are executed by the parser as the SQL/NF query statements are parsed and grammar rules are recognized.

The SQL_QUERY node (Figure 4.10) contains an identifier to indicate what type of query expression it is (NEST, UNNEST, QUERY_SPEC = SFW query expression , or FUNC_QE = query expression with a function), the function name associated with the query expression if type = FUNC_QE, a pointer to the first SELECT_NODE of a select node list, a pointer to the first FROM_NODE of a from node list, a pointer to the first WHERE_NODE of a where node list, and the new column name of an *AS* clause. If select_list = NULL then the query expression is a "*SELECT ALL FROM...*" query. If the where_list = NULL then the query has no *WHERE* clause. When type = (NEST or UNNEST) the FROM_NODE contains the table-name and the

SELECT_NODE
type
attr
reference
nest
next

Figure 4.11. The SELECT_NODE structure

FROM_NODE
type
name
nest
next

Figure 4.12. The FROM_NODE structure

SELECT_NODE contains the column-list.

The SELECT_NODE (Figure 4.11) contains an identifier to indicate what type of element in the *SELECT* clause it is (COLUMN_SPEC = attribute name, or QUERY_EXP = nested query expression), a pointer to an ATTRDESC data structure[1] which holds the attribute name if type = COLUMN_SPEC, the reference name (eg. ref-name.attr-name) if one is given, a pointer to an SQL_QUERY node if type = QUERY_EXP, and a pointer to a SELECT_NODE which contains the next element of the *SELECT* clause.

The FROM_NODE (Figure 4.12) contains an identifier to indicate what type of element in the *FROM* clause it is (TABLE_NAME or QUERY_EXP), the name of the table if type = TABLE_NAME, a pointer to an SQL_QUERY node if type = QUERY_EXP, and a pointer to a FROM_NODE which contains the next element of the *FROM* clause.

The WHERE_NODE (Figure 4.13) contains an identifier to indicate what type of element in the *WHERE* clause it is (PRED = predicate, or QUERY_EXP), a pointer to a PRED_NODE data structure[1] which contains the predicate information if type = PRED, a pointer to an SQL_QUERY

[1]This data structure is used by the Colby algebra tree and defined in Section 4.5.

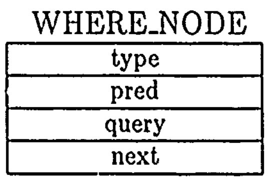

Figure 4.13. The WHERE_NODE struture

node if type = QUERY_EXP, and a pointer to a WHERE_NODE which contains the next element of the *WHERE* clause.

Now to see how it is all put together, consider the following query statement:

```
SELECT DNO, NAME
FROM   DEPT, EMP
WHERE DNAME = "MKT" AND AGE > 30
         AND DEPT.DNO = EMP.DNO;
```

The query tree in Figure 4.14 is the result of parsing the above SQL/NF query. The empty blocks in the ATTRDESC and PRED_NODE data structures are filled in when converting the SQL/NF query tree to the Colby algebra tree. To see how a nested query statement in the *SELECT* clause would look in an SQL/NF query tree, consider the following query statement:

```
SELECT (SELECT SAL
          FROM   EMP
          WHERE ENAME = "SMITH")
FROM   COMP;
```

The query tree in Figure 4.15 is the result of parsing the above SQL/NF query. The nested query in the *SELECT* clause is identified in the query tree by the "nest" element of the top level SELECT_NODE which is of the type QUERY_EXP. The empty blocks in the ATTRDESC and PRED_NODE data structures are filled in when converting the SQL/NF query tree to the Colby algebra tree.

The BNF definitions for the structured queries (NEST and UNNEST), require an additional data structure be used, in conjunction with the parser, for holding the information of a "column list" as shown in Figure 4.16. The COL_SPEC_NODE contains a column-name, a reference-name if one

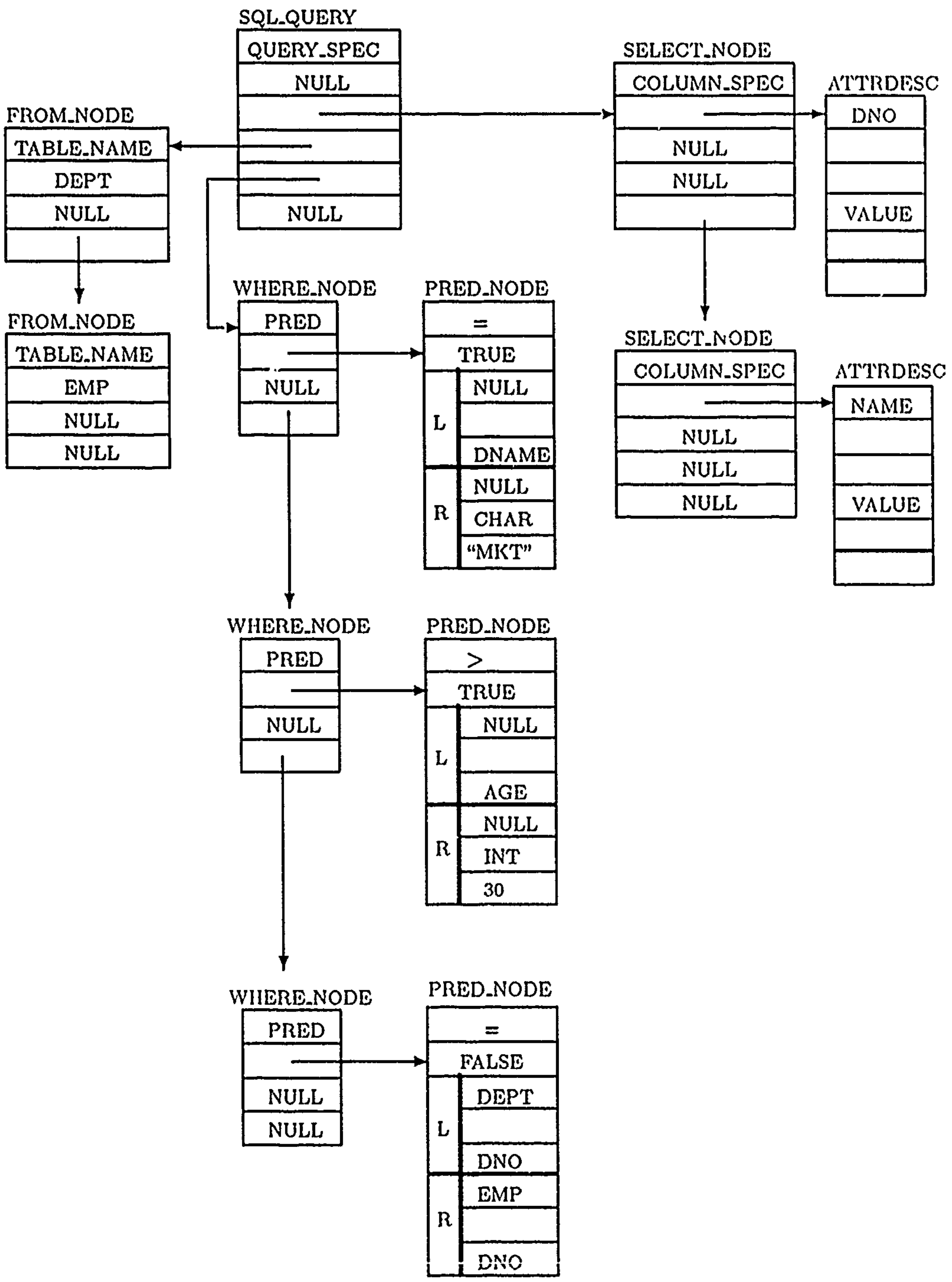

Figure 4.14. SQL/NF Query Tree

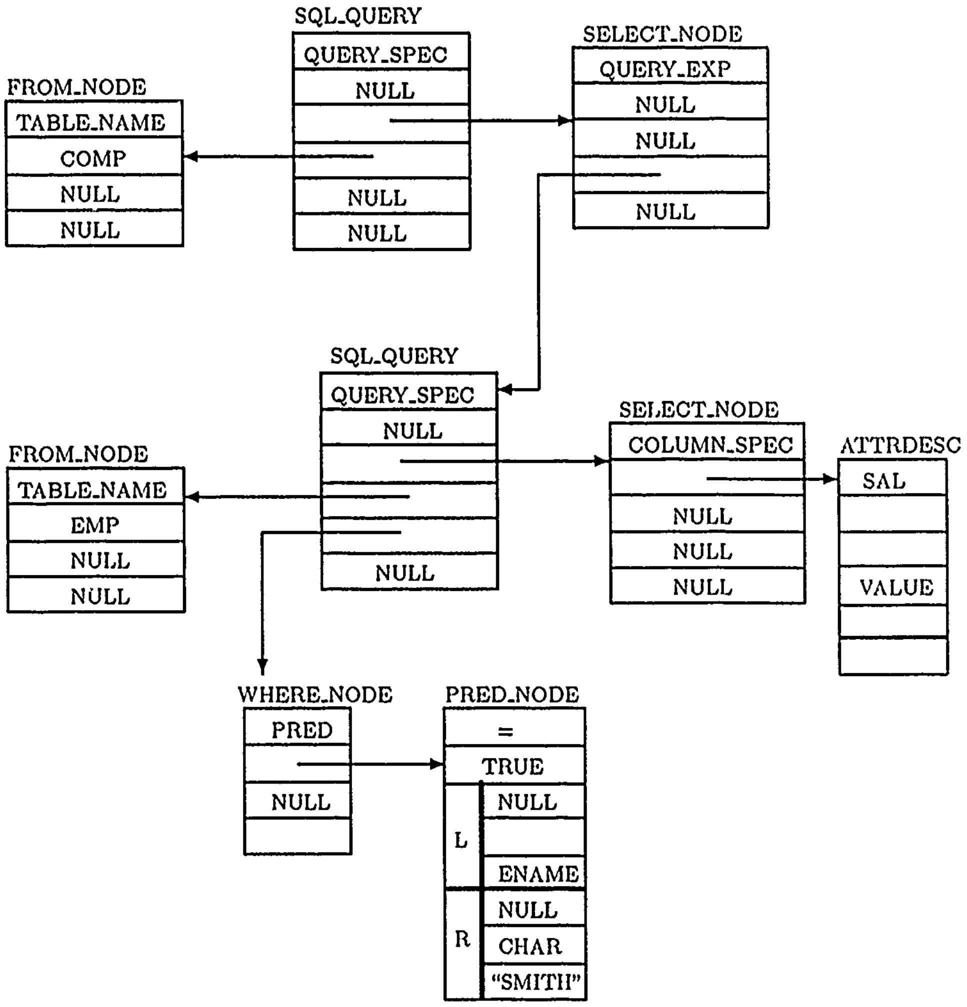

Figure 4.15. SQL/NF nested Query Tree

name
ref_name
next

Figure 4.16. The COL_SPEC_NODE structure

QUERY

OPERATOR	
argument	
input[LEFT]	input[RIGHT]

Figure 4.17. The QUERY node structure

is used, and a pointer to a COL_SPEC_NODE which is the next element in the column list. Once the column_list structure is complete, it is translated into a select_list as part of a SQL_QUERY node.

4.5 Colby Algebra Query Tree process

The basic structure of the Query tree was established as part of the Thesis effort by Mankus(10), with some additions made to several of the tree components. The tree consists of a series of QUERY nodes linked together, along with supporting data structures, to logically provide an access procedure to a database with respect to the relational operators of the Colby Algebra. Each QUERY node represents a relational operator in the Colby relational algebra.

The QUERY node (Figure 4.17) contains an identifier to indicate the operator (SELECT, PROJECT, PRODUCT, NEST, UNNEST) for the query expression, a nested ARGUMENT structure, and a pair of pointers to additional input QUERY nodes which make up the tree structure.

ARGUMENT
name
reltype
new_name
pred
list

Figure 4.18. The ARGUMENT structure

LIST
attr
cond
sublist
next

Figure 4.19. The LIST node structure

The input QUERY nodes identified with LEFT and RIGHT are both required if the OPERATOR = PRODUCT, because a Cartesian-product is a binary operation. Only the LEFT input QUERY node is used for the remaining operators identified earlier. When other binary operators (UNION, DIFFERENCE, INTERSECTION) are implemented, both input QUERY nodes will also be required.

The ARGUMENT structure (Figure 4.18) contains information for the QUERY node operator and consists of the relation name, the relation scheme type, the new name of the subrelation for the NEST operator, a pointer to a PRED_NODE which contains selection condition information, and a pointer to a LIST node which is the first node of the attribute list. Depending on the QUERY node operator and location in the Query tree, different elements in the structure will or will not be used.

The LIST node (Figure 4.19) is used to maintain information of attributes to be projected or navigated across to reach nested attributes. The node is used to construct the Select List, Project List, Nest List, and Unnest List for the Colby Algebra operators. It consists of a pointer to an ATTRDESC structure which contains information about the attribute, a pointer to a PRED_NODE

ATTRDESC

name	
type	
size	
value	u_flag
	data_type
rvatype	
parentrel	

Figure 4.20. The ATTRDESC structure

PRED_NODE

oper	
constant_on_right	
left	ref_name
	u_flag
	op_type
right	ref_name
	u_flag
	op_type

Figure 4.21. The PRED_NODE structure

structure which contains condition criteria if the attribute is a relation valued attribute, a pointer to a LIST node which identifies a nested list of attributes for the relation valued attribute of the current LIST node, and a pointer to a LIST node which is the next item in the List for the QUERY node.

The ATTRDESC structure (Figure 4.20) contains all the information on an attribute. It consists of the name of the attribute, the domain of the attribute (CHAR, INT, FLOAT, PREV_DEFINED), the size of the attribute identifies the number of bytes (CHAR, INT, FLOAT) or the number of nested attributes (PREV_DEFINED), a nested VALUE structure that is used for storing tuple information[2], the scheme-name for a relation valued attribute, and the name of the parent relation for the attribute.

The PRED_NODE contains the information pertaining to a selection condition or predicate

[2]This structure is explained further in Section 4.6.

for the associated ARGUMENT or LIST node. It consists of an identifier to indicate the operator for the node ($=, != , <, >, \leq, \geq$, NOT, AND, OR), an identifier to indicate if the right hand side of the predicate contains a constant value (eg. "SMITH", 123), and two operand nested structures (left and right) which correspond to the left and right side of the predicate. The operand structure consists of the reference name for the attribute used for that side of the predicate, a flag to indicate what type of value is used for that side of the predicate (PRED, CHAR, INT, FLOAT), and a nested *union* structure which contains the actual value of the type identified by the flag. The PRED type value is used when the operator for the predicate is one which links two predicates at the current level in the Query tree together such as NOT, AND, or OR.

4.5.1 SQL/NF to Colby Algebra translation. The three clauses (*SELECT, FROM, WHERE*) of an SQL/NF query[3] can be translated into a Colby Algebra query using the relational operators[4] (select, project, and cartesian-product) with little difficulty. The attribute-list of the *SELECT* clause corresponds to the project-list of the Projection operator. The predicate-list of the *WHERE* clause corresponds directly to the Condition of the Selection operator. The relation-list of the *FROM* clause corresponds to the Relations of the Cartesian-product operator. The attribute-list of the SQL/NF *NEST* clause corresponds to the Nest-list of the Colby Nest operator, and the same for the *UNNEST* clause and Nest operator. When a nested SFW-expression appears in the *SELECT* clause, the relation-list of the nested *FROM* clause along with the attribute-list of the nested *SELECT* clause provides the path and attributes for the project-list, and the relation-list provides the path for the select-list. The nested select-list condition is obtained from the nested *WHERE* clause.

The translation of the SQL/NF query tree in Figure 4.14 which was generated from the following SQL/NF query statement:

[3]See Section 3.2.2
[4]See Appendix B

```
SELECT DNO, NAME
FROM   DEPT, EMP
WHERE  DNAME = "MKT" AND AGE > 30
       AND DEPT.DNO = EMP.DNO;
```

Would produce the Colby Algebra query tree in Figure 4.22, and represents the following Colby Algebra query:

$$\pi((DNO, NAME)\,\sigma(\times(\sigma(DEPT_{DNAME = \text{"MKT"}}), \sigma(EMP_{AGE > 30}))_{DEPT.DNO = EMP.DNO}))$$

The translation of the SQL/NF query tree in Figure 4.15 which was generated from the following SQL/NF query statement:

```
SELECT (SELECT SAL
        FROM   EMP
        WHERE  ENAME = "SMITH")
FROM   COMP;
```

Would produce the Colby Algebra query tree in Figure 4.23, and represents the following Colby Algebra query:

$$\pi((EMP(SAL))\,\sigma(COMP(EMP_{ENAME = \text{"SMITH"}})))$$

The translation process has been given the additional responsibility of performing error checking on the input query statement. All attribute names used in the *SELECT* and *WHERE* clauses are checked to verify the correct parent relation name exists within the associated *FROM* clause. The verification is accomplished by checking the symbol table. When the attribute in the SQL/NF query tree is matched to a relation name then the attribute is added to the Colby query tree along with the information required for the ATTRDESC. Once all the relation names of the *FROM* clause have been used for attribute testing, a final check is made to see if all attributes have been translated to the Colby query tree, if not, an error condition is set.

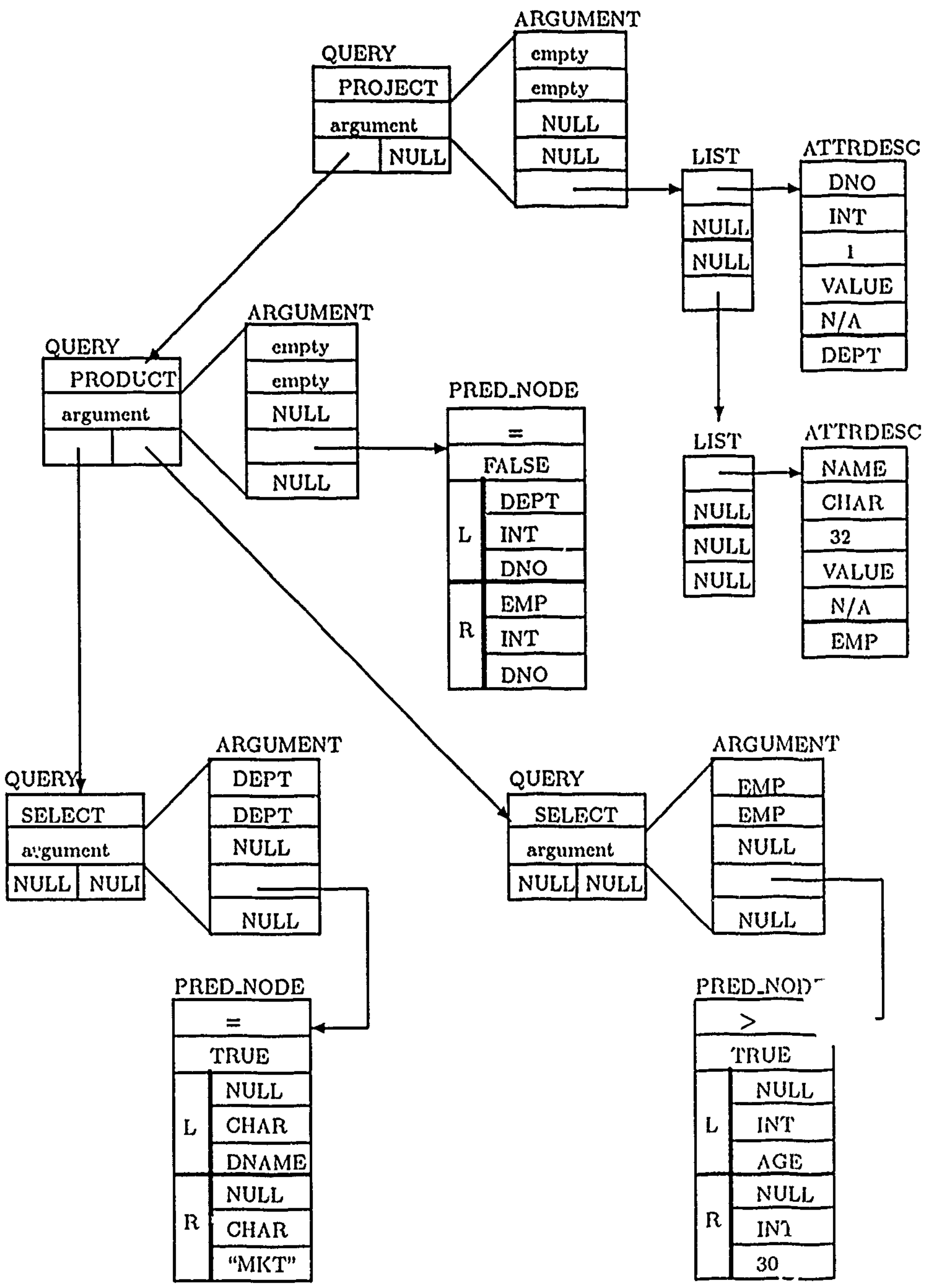

Figure 4.22. Sample Colby Query Tree

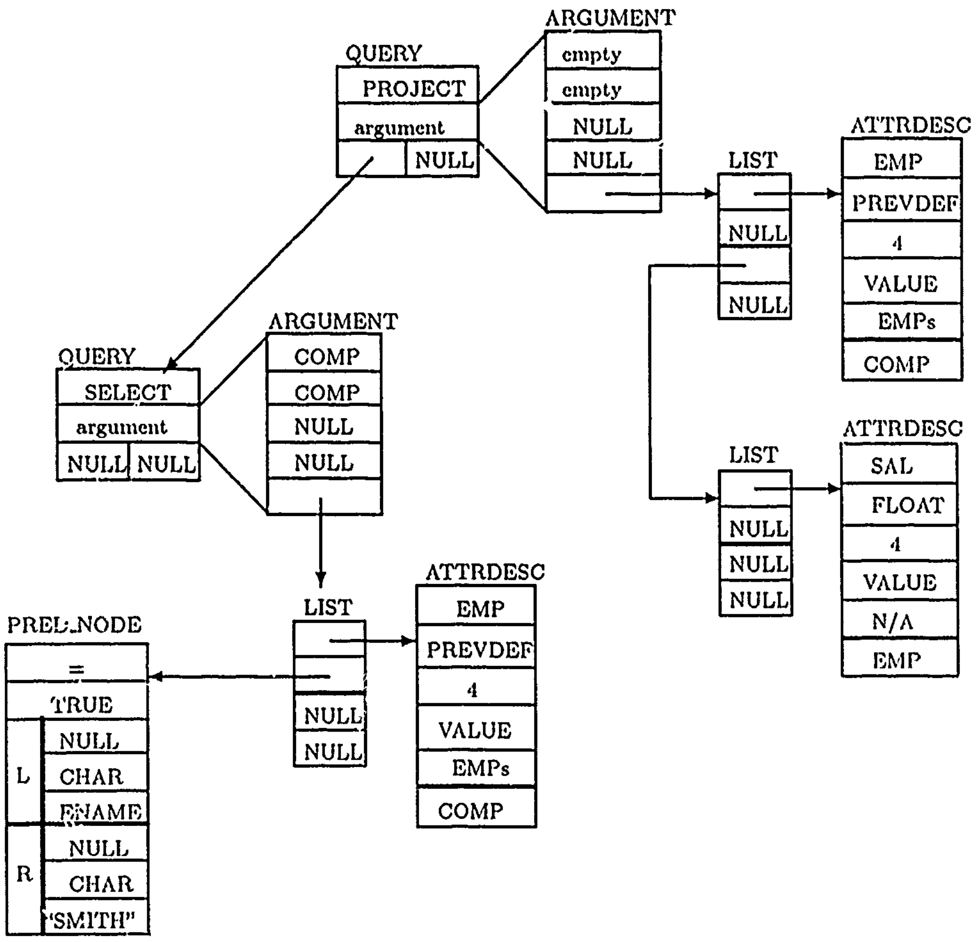

Figure 4 23. Nested SFW-expression in Query Tree

name
type
index
numb
parent_name
parent_index
next

Figure 4.24. The NEST_TABLE_NODE structure

4.5.2 NEST and UNNEST Process. The *NEST* and *UNNEST* clauses present a problem when trying to match attributes and their parent relation by checking the symbol table. This is due to the nature of the commands to restructure the relation. The solution to this problem brought about the requirement for a temporary symbol table to contain the new relation valued attributes and the nested attributes new parent information.

The NEST_TABLE_NODE structure (Figure 4.24) is used to provide this temporary symbol table. It consists of the name of the attribute, what operator type (NEST or UNNEST) has changed the parent of the attribute, the symbol table INDEX value for the attribute, the symbol table NUMB value for the attribute, the new parent name for the attribute, the new parent INDEX value from the symbol table, and a pointer to the next NEST_TABLE_NODE in the temporary symbol table. When the attribute is a relation valued attribute created by the *NEST* clause the "numb" value is the number of attributes in the *ON* clause.

When a *NEST* or *UNNEST* clause is encountered in the SQL/NF query the temporary symbol table is created. The error checking process takes place during the creation of the temporary symbol table. Now when an attribute is not found in the permanent symbol table, the temporary symbol table is checked and the ATTRDESC information is provided if the attribute is found.

Let us consider the following query using the *NEST* clause on the COMP relation of Figure 4.25.

COMP

DNO	DNAME	ENO	SAL	ENAME

Figure 4.25. COMP relation table

```
SELECT DNO, (SELECT ENO
             FROM   EMP)
FROM   (NEST ENO, SAL, ENAME
        ON    COMP AS EMP);
```

The Colby query tree generated by the above SQL/NF query is shown in Figure 4.26 and translates to the following Colby Algebra query:

$$\pi((DNO, EMP(ENO))\,\nu(COMP(ENO, SAL, ENAME) \rightarrow EMP))$$

The *UNNEST* clause is translated into a query tree almost identical to the NEST operator query tree, with the exception being no new_name is assigned in the ARGUMENT.

4.6 Extensions to the Colby Query Tree

In order to provide for non-relational algebra database operations to be passed on to the Query Optimizer and subsequent stages in the EXODUS architecture (see Figure 2.5), several additional values for the OPERATOR element of the QUERY node were defined. These extensions reflect the operations of the DDL and DML statements.

When creating or dropping a relation table via a *CREATE TABLE* or *DROP TABLE* command, the EXODUS storage manager must be informed of the operation to allocate or deallocate the relation table. The basic structure of the Colby query tree for these operations is a single QUERY node with the name of the relation in the ARGUMENT and an OPERATOR value of CREATE_REL for *CREATE TABLE* or DROP_REL for *DROP TABLE* .

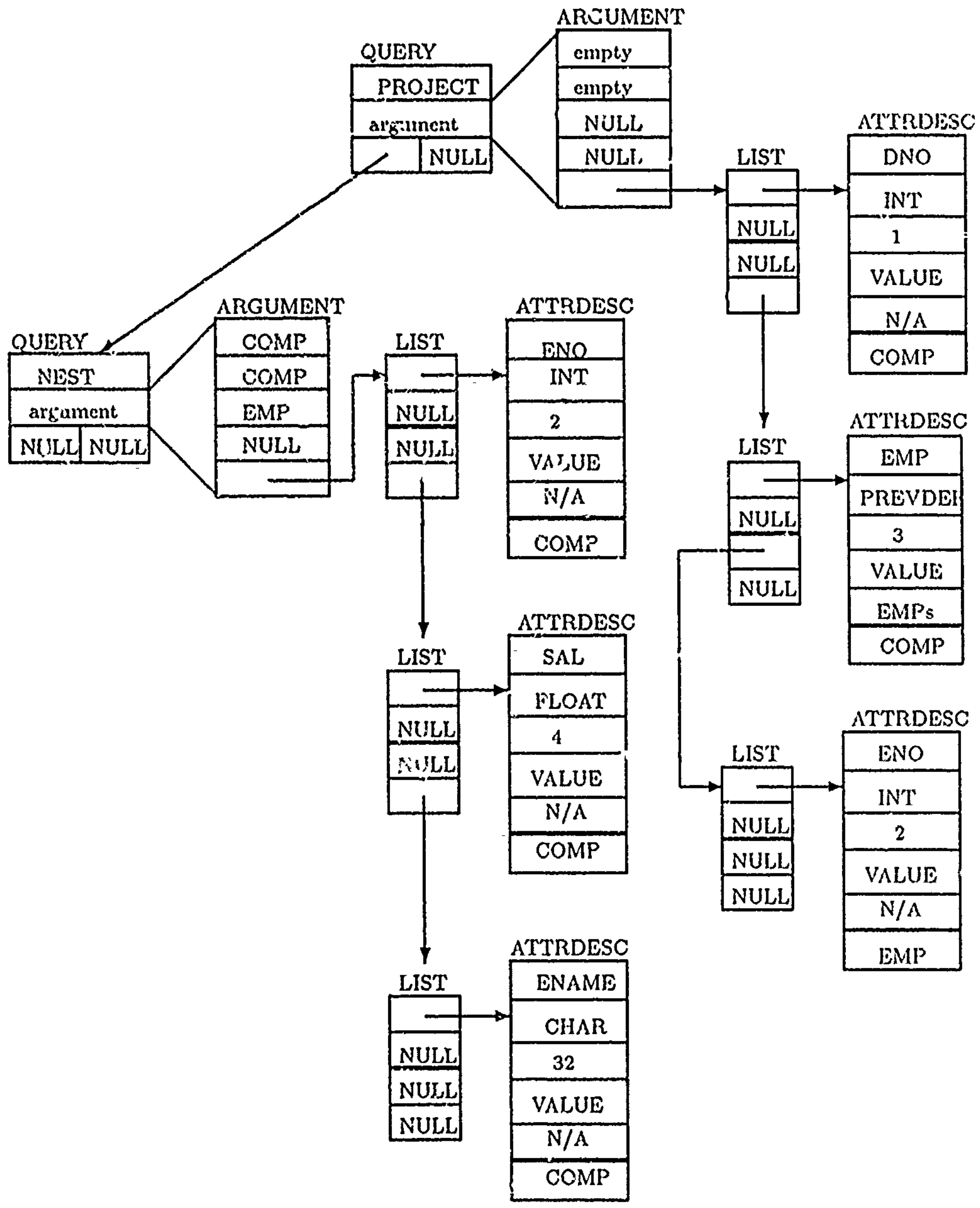

Figure 4.26. NEST operator Query Tree

EMP

ENO	ENAME	CHILDREN	
		CNAME	DOB

Figure 4.27. The EMP relation table

A major extension that was developed but not implemented at this time, concerns the DML commands of *STORE, MODIFY,* and *ERASE.* This requirement led to the modification of the ATTRDESC structure (Figure 4.20) to include a way to store tuple information to be applied to the database. The modification as the addition of the "value" nested structure which is similar to the operand structure used in the PRED_NODE (Figure 4.21). The value structure consists of a flag to indicate what type of value is used for that attribute (CHAR, INT, FLOAT), and a nested *union* structure which contains the actual value of the type identified by the flag. For example, if we add a tuple to the EMP relation in Figure 4.27 using the *STORE* command with the following SQL/NF command:

```
STORE EMP (ENO, ENAME, CHILDREN(CNAME, DOB))
        VALUES <123, SMITH, (<STEVE, MAY>) >;
```

The Colby query tree structure generated by the above *STORE* command can be seen in Figure 4.28. Multiple tuples for the nested relation(s) (eg. CHILDREN) are identified in the query tree by adding an additional list node (via next) with the ATTRDESC reflecting the same relation valued attribute name (CHILDREN) and another sublist with the tuple data. When translating the SQL/NF query trees for the *MODIFY* and *ERASE* commands, the Colby query trees are constructed in a similar manner.

4.7 Testing and Validation

Upon completion of implementing the Data Dictionary process several relations were created and deleted (1NF relations, and Nested relations with and without scheme definitions), to test all

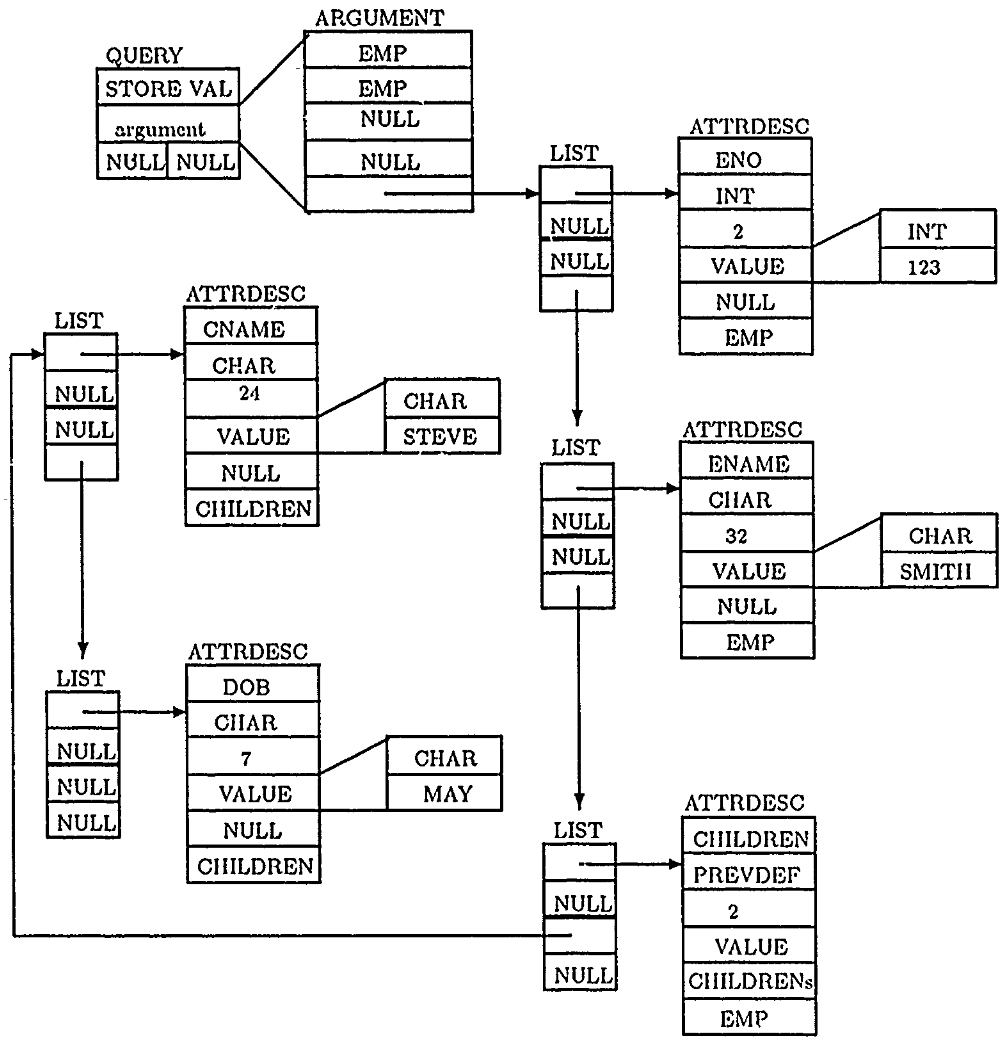

Figure 4.28. STORE_VALUE QUERY Tree

the combinations of the DDL statements that were implemented in the parser, some of the results were used in Section 4.3.

After implementing the translator, several queries were input to the parser. The Colby query tree was built for all combinations of the SQL/NF SFW-expressions that were implemented in the parser. These included SFW queries on top level attributes as well as nested attributes, queries on multiple relations, and queries utilizing the NEST and UNNEST operators. In order to verify the QUERY tree structure a set of print procedures were developed to provide a formatted output of the Colby query tree. Some sample test cases and their results are provided in Appendix D.

4.7.1 Translator limitations. Testing the capabilities of the translator was limited to a subset of the Query facilities and DDL statements defined in Appendix C.

The Query facilities implemented only include <query spec>, <structured query>, and <nested query expression>. The <structured query> capabilities are limited to NEST and UNNEST. The <query spec> capabilities do not include the *PRESERVE* clause and <predicate> is limited to <comparison predicate>.

The DDL statements implemented include <schema>, <scheme>, and <drop statement>. These statements are limited by not including any capabilities dealing with constraints.

V. Conclusion

5.1 Summary

This thesis effort accomplished several objectives, resulting in the design and implementation of an SQL/NF to Colby Algebra query translator as part of the Triton system using the EXODUS tool kit. The main objectives included developing:

1. A parser to recognize SQL/NF statements and execute associated processes.
2. A persistent data dictionary and the associated processes to create and maintain it.
3. A query tree structure in the form of the SQL/NF query expressions.
4. A translation process to convert the SQL/NF query tree into the equivalent Colby relational algebra query tree.
5. A process to walk down the Colby query tree and display the contents at each node.

The parser was implemented using the UNIX tools of LEX and YACC. The key words in the BNF for SQL/NF were defined in the lexical analyzer created by LEX and the BNF definitions were translated into the appropriate grammar definitions for YACC. Print statements were added to each grammar rule to show the parsing process for recognizing the components of the SQL/NF statements.

The composition of the Data Dictionary we used was developed as part of the thesis work by Mankus(10). This design took advantage of the persistence feature of EXODUS which allowed the data dictionary to remain in the storage manager between program executions. The data structures and associated processes required to interpret and implement the SQL/NF DDL statements for use with the data dictionary were developed. These processes are called by the parser as the DDL statement is recognized.

A query tree structure was developed to represent the structure of the SQL/NF query statement. Each query node consisted of three main elements which reflect the select, from, and where clauses of the SQL/NF query statement. The processes were developed to build the query tree as the parser recognized the components of each of the statement clauses. The design included providing for nested queries within any one of the clauses. Only a subset of the SQL/NF query statements were implemented, mainly the ones that can be directly translated into the Colby relational algebra.

The SQL/NF to Colby Algebra translation process was developed using the SQL/NF query tree as input and producing the equivalent relational algebra query tree as output. The query nodes and the tree structure resemble a general structure required by the query optimizer stage in the EXODUS architecture. The query tree structure for translating DML statements was designed but not implemented. One of the main tasks of the translator process is to check the components of the SQL/NF query tree for legal queries by matching relations and their attributes identified in the query statement. This matching process is accomplished via procedures developed for checking the contents of the data dictionary.

As part of the testing process to verify the composition of the Colby query tree a group of print procedures were developed to walk down the tree and output each of the elements of the data structures for each query node. These print procedures were developed so as to correspond directly with each of the data structures associated with the QUERY node.

5.2 Future Recommendations

Enhancements to the current system would begin with continuing the implementation of the DML commands as to enable manipulation of the data in the database. This could be followed by implementing the set operations (UNION, INTERSECTION, DIFFERENCE) between query expressions. The next major enhancement to the system would be to extend the relational algebra to be able to handle query expressions with functions and the predicates of the *WHERE* clauses "search condition" not implemented in the current system. Finally, enhancements to the Data Dictionary would include the addition of table constraint definitions and column constraint specifications. Serious thought should be given to changing the symbol table into a version which utilizes the ITEM and Table data structures (Figure 4.4) and is also persistent.

Appendix A. *YACC and LEX formats*

A.1 YACC

The general format for a YACC specification is:

```
{declarations}

%%

{rules}

%%

{programs}
```

where one or more sections can be omitted except the first %%. The rules section contains lines of the type:

```
name: body;
```

where the colon and semicolon are YACC punctuation, *name* is a nonterminal symbol and *body* is a sequence of zero or more names and literals (A literal consists of characters enclosed in single quotes.) Nonterminal 'names' are declared in the declaration section as:

```
%token name1 name2 ...
```

Grammar rules which have the same left hand side can be rewritten using the vertical bar "|" instead of rewriting the left hand side, for example:

```
name: surname;

name: firstname surname;

name: firstname middlename surname;
```

can be given to YACC in the format identified in Figure A.1 where *surname, firstname* and *middlename* are literals. The statements enclosed by { and } are the actions that are executed once the associated rule is satisfied. In each case a different variation of the build_name() procedure is called for each rule.

```
name: surname
      {
         build_name(NULL, NULL, string1);
       }
    | firstname surname
      {
         build_name(string1, NULL, string2);
      }
    | firstname middlename surname
      {
         build_name(string1, string2, string3);
      }
    ;
```

Figure A.1. Example YACC rule

A.2 LEX

Like YACC, the general format for a LEX program is:

{definitions}

%%

{rules}

%%

{user defined subroutines}

where the definitions and user defined subroutines are optional.

Each regular expression represents the user's control decision. It is written in the form of a table, with the regular expressions on the left and LEX actions to the right. The normal C escapes, like \$t$, and \$n$, are recognized and the back-slash can be used to escape LEX operators For example, if the user wants to recognize key words of the SQL/NF query language, the following LEX rules would be appropriate:

```
"SELECT" return(T_SELECT);

"FROM" return(T_FROM);

"WHERE" return(T_WHERE);
```

The actions taken are the return of a "token" value for each key word.

Appendix B. *Colby Algebra Definitions*

B.1 Query Formats

1. Selection operator:	$\sigma(Relation_{Condition}(Select\ List))$
2. Projection operator:	$\pi((Project\ List)\ Relation)$
3. Nest operator:	$\nu(Relation\ (Nest\ list) \rightarrow New\ Subrelation)$
4. Unnest operator:	$\mu(Relation(Unnest\ list))$
5. Cartesian-product operator:	$\times(Relation1(JoinPath), Relation2)$
6. Join operator:	$\bowtie(Relation1(Join\ Path), Relation2)$
7. Union operator:	$\cup^e(REL1, REL2)$
8. Difference operator:	$-^e(REL1, REL2)$
9. Intersection operator:	$\cap^e(REL1, REL2)$

B.2 Selection (σ).

Let R be a relation scheme.

Then, L is a 'select list' of R if:

1. L is empty.
2. L is of the form $(R_{1c_1}L_1, R_{2c_2}L_2, \ldots, R_{nc_n}L_n)$ $(1 \leq n \leq |\ RAttr(R)\ |)$ where each R_i is a relation-valued attribute of R, c_i is a condition on R_i, and L_i is a select list of R_i.

Let r be a relation with relation scheme R.

1. $\sigma(r_c) = \{t \in r \mid c(t) = true\}$

2. $\sigma(r_c(R_{1c_1}L_1, R_{2c_2}L_2, \ldots, R_{nc_n}L_n)) =$

$$\begin{aligned}
&\{t \mid \exists t_r \in r \mid \\
&\quad (t[Attr(R) - \{R_1, R_2, \ldots, R_n\}] = t_r[Attr(R) - \{R_1, R_2, \ldots, R_n\}]) \\
&\quad \wedge (c(t_r) = true) \\
&\quad \wedge (t[R_1] = \sigma((t_r[R_1])_{c_1} L_1) \neq \emptyset) \\
&\quad\quad \vdots \\
&\quad \wedge (t[R_n] = \sigma((tr[R_n])_{c_n} L_n) \neq \emptyset)\}
\end{aligned}$$

where $L = (R_{1c_1}L_1, R_{2c_2}L_2, \ldots, R_{nc_n}L_n)$ is a select list of ' ssibly empty) select lists of R_i's and c is a condition on R.

B.3 Projection (π).

Let R be a relation scheme.

Then, L is a 'project list' of R if:

1. L is empty.

2. L is of the form $(R_1L_1, R_2L_2, \ldots, R_nL_n)$ where each R_i is an attribute of R and L_i is a project list of R_i (L_i is empty if R_i is an atomic-attribute).

Let r be a relation with relation scheme R.

1. $\pi(r) = r$.

2. $\pi((R_1L_1, R_2L_2, \ldots, R_nL_n)r) =$

$$\{t \mid (\exists r \in r) \wedge (t[R_1] = f(t_r, R_1L_1)) \wedge \ldots \wedge (t[R_n] = f(t_r, R_nL_n))\}$$

where
$$\begin{aligned}
f(t_r, R_iL_i) &= t_r[R_i] \; if \; R_i \in FAttr(R) \\
&= \pi(L_i(t_r[R_i])) \; if \; R_i \in RAttr(R)
\end{aligned}$$

where $(R_1L_1, R_2L_2, \ldots, R_nL_n)$ is a *project list* of R.

B.4 Nest (ν).

Let R be a relation scheme.

Then, L is a 'nest list' of R if:

1. L is of the form $(R_1, \ldots, R_n)$ where each $R_i \in Attr(R)$.

2. L is of the form $(R_i L_i)$ where $R_i \in RAttr(R)$ and L_i is a nest list of R_i.

Let r be a relation with relation scheme R and let A be a new attribute name such that $A \ni Attr(R)$.

1. $$\begin{aligned}\nu(r(R_1, \ldots, R_n) \to A) = \{t \mid \exists t_r \in r \mid\ & (t[Attr(R) - \{R_1, \ldots, R_n\}] = t_r[Attr(R) - \{R_1, \ldots, R_n\}]) \wedge \\ & (t[A] = \{s[R_1, \ldots, R_n] \mid s \in r \mid \\ & \quad (s[Attr(R) - \{R_1, \ldots, R_n\}] = t_r[Attr(R) - \{R_1, \ldots, R_n\}])\})\}\end{aligned}$$

2. $$\begin{aligned}\nu(r(R_i L_i) \to A) = \{t \mid \exists t_r \in r \mid\ & (t[Attr(R) - \{R_i\}] = t_r[Attr(R) - \{R_i\}]) \\ & \wedge(t[R_i] = \nu(t_r[R_i]L_i \to A))\}\end{aligned}$$

B.5 Unnest (μ).

Let R be a relation scheme.

Then, L is an 'unnest list' of R if:

1. L is of the form (R_i) where $R_i \in RAttr(R)$.

2. L is of the form $(R_i L_i)$ where $R_i \in RAttr(R)$ and L_i is an unnest list of R_i.

Let r be a relation with relation scheme R.

1. $\mu(r(R_i)) = \{t \mid \exists t_r \in r \mid$
$(t[Attr(R) - \{R_i\}] = t_r[Attr(R) - \{R_i\}])$
$\wedge(t[Attr(R_i)] \in t_r[R_i])\}$

2. $\mu(r(R_iL_i)) = \{t \mid \exists t_r \in r \mid$
$(t[Attr(R) - \{R_i\}] = tr[Attr(R) - \{R_i\}])$
$\wedge(t[R_i] = \mu(t_r[R_i]L_i))\}$

B.6 Join Path.

Let R be a relation scheme.

L is a "join path" of R if:

1. L is empty.

2. L is of the form (R_iL_i) where R_i is a relation-valued attribute of R ($R_i \in RAttr(R)$) and L_i is a join path of R_i.

B.7 Cartesian-Product ($\times$).

Let r and q be two relations with relation schemes R and Q respectively.

1. $\times(r,q) \;=\; \{t \mid \exists t_r \in r, \exists t_q \in q \mid$

   $(t[Attr(R)] \;=\; t_r) \wedge (t[Attr(Q)] \;=\; t_q)\}$

2. $\times(r(R_iL_i),q) \;=\; \{t \mid \exists t_r \in r \mid$

   $(t[R_i] \;=\; \times(t_r[R_i]L_i, q)$

   $\wedge(t[Attr(R) - \{R_i\}] \;=\; t_r[Attr(R) - \{R_i\}])\}$

   where (R_iL_i) is a join path of R.

We assume that common attributes in R and Q are renamed in order to resolve ambiguity.

B.8 Join ($\bowtie$).

Let r and q be two relations with relation schemes R and Q respectively and let L be a join path of R.

1. $\bowtie(r,q) \;=\; \{t \mid \exists t_r \in r, \exists t_q \in q \mid$

   $(t[R_1,\ldots,R_n] \;=\; t_r[R_1,\ldots,R_n] \;=\; t_q[R_1,\ldots,R_n])$

   $\wedge(t[Attr(R) - \{R_1,\ldots,R_n\}] \;=\; t_r[Attr(R) - \{R_1,\ldots,R_n\}])$

   $\wedge(t[Attr(Q) - \{R_1,\ldots,R_n\}] \;=\; t_r[Attr(Q) - \{R_1,\ldots,R_n\}])$

   where $\{R_1,\ldots,R_n\}$ are the common attributes of R and Q.

2. $\bowtie(r(R_iL_i),q) \;=\; \{t \mid \exists t_r \in r \mid$

   $(t[R_i] \;= \bowtie(t_r[R_i]L_i, q) \neq \emptyset)$

   $\wedge(t[Attr(R) - \{R_i\}] \;=\; t_r[Attr(R) - \{R_i\}])\}$

B.9 *Extended Binary Operators.*

The following applies to all the binary operators:

1. Let r1 and r2 be two relations with relation scheme R.
2. Let k(R) = key(R) ∪ FAttr(R) and let m(R) = RAttr(R) - k(R), where key(R) is the set of attributes which determine the key for R.

B.9.1 *Union* $(\cup, \cup^e)$.

1. $\cup(r_1, r_2) \;=\; \{t \mid (t \in r_1) \vee (t \in r2)\}$.
2. $$\begin{aligned}\cup^e(r_1, r_2) \;=\; \{t \mid ((t \in r1) \wedge (\forall t_2 \in r_2, t_2[k(R)] \neq t[k(R)]))\\ \vee(t \in r_2) \wedge (\forall t_1 \in r_1, t_1[k(R)] \neq t[k(R)]))\\ \vee(\exists t_1 \in r_1, \exists t_2 \in r_2 \mid\\ (t[k(R)] \;=\; t_1[k(R)] \;=\; t_2[k(R)])\\ \wedge(t[R_1] \;=\; \cup^e(t_1[R_1], t_2[R_1]))\\ \vdots\\ \wedge(t[R_l] \;=\; \cup^e(t_1[R_l], t_2[R_l]))))\}\end{aligned}$$

where $R_i \in m(R)$ $(1 \leq i \leq l)$.

B.9.2 *Difference* $(-, -^e)$.

1. $-(r_1, r_2) \;=\; \{t \mid (t \in r_1) \vee (t \ni r2)\}$.
2. $$\begin{aligned}-^e(r_1, r_2) \;=\; \{t \mid ((t \in r1) \wedge (\forall t_2 \in r_2, t_2[k(R)] \neq t[k(R)]))\\ \vee(\exists t_1 \in r_1, \exists t_2 \in r_2 \mid\\ (t[k(R)] \;=\; t_1[k(R)] \;=\; t_2[k(R)]) \wedge (t1 \neq t2)\\ \wedge(t[R_1] \;=\; -^e(t_1[R_1], t_2[R_1]))\\ \vdots\\ \wedge(t[R_l] \;=\; -^e(t_1[R_l], t_2[R_l]))))\}\end{aligned}$$

where $R_i \in m(R)$ $(1 \leq i \leq k)$

B.9.3 Intersection $(\cap, \cap^e)$.

1. $\cap(r_1, r_2) = \{t \mid (t \in r1) \wedge (t \in r_2)\}$.

2. $$\begin{aligned} \cap^e(r_1, r_2) = \{t \mid (\exists t_1 \in r_1, \exists t_2 \in r_2 \mid \\ (t[k(R)] &= t_1[k(R)] = t_2[k(R)]) \\ \wedge(t[R_1] &= \cap^e(t_1[R_1], t_2[R_1])) \\ &\vdots \\ \wedge(t[R_l] &= \cap^e(t_1[R_l], t_2[R_l]))))\} \end{aligned}$$

where $R_i \in m(R)$ $(1 \leq i \leq l)$

Appendix C. *SQL/NF BNF*

The following is a modified BNF definition of the Query Facilities, Data Manipulation Language (DML), and Data Definition Language (DDL) for SQL/NF. The original version is found in (13), which used RDL (1) as the baseline definition. A TERMINAL symbol (key word) is identified as a word consisting of all capital letters. Non-distinguished symbols are enclosed with "⟨ ⟩". The structure "[ ]" indicates an optional entry and the structure "· · ·" indicates an additional zero or more repetitions of the previous entry. Braces are used for grouping in the BNF. Except where modified by braces, sequencing has precedence over disjunction (indicated by "|").

C.1 Query facilities

- ⟨query expression⟩ ::= ⟨query spec⟩ | ⟨structured query⟩ | ⟨function⟩ (⟨query expression⟩)
  | ⟨nested query expression⟩ | ⟨query expression⟩ ⟨set operator⟩ ⟨query expression⟩
- ⟨structured query⟩ ::= NEST ⟨nested query expression⟩ ON ⟨column list⟩ [AS ⟨column name⟩]
  | UNNEST ⟨nested query expression⟩ ON ⟨column list⟩
  | ORDER ⟨nested query expression⟩ BY ⟨sort spec⟩ · · ·
- ⟨sort spec⟩ ::= {⟨unsigned integer⟩ | ⟨column name⟩} [ASC | DESC]
- ⟨query spec⟩ ::= ⟨select from spec⟩ [WHERE ⟨search condition⟩ [PRESERVE ⟨table list⟩]]
- ⟨select from spec⟩ ::= SELECT ⟨select list⟩ FROM ⟨table list⟩ | ⟨table name⟩
- ⟨select list⟩ ::= ALL | ⟨select spec list⟩
- ⟨select spec list⟩ ::= ⟨select spec⟩ [{,⟨select spec⟩} · · ·]
- ⟨select spec⟩ ::= ⟨column expression⟩ | ⟨reference name⟩.ALL
- ⟨column expression⟩ ::= ⟨value expression⟩ [AS ⟨column name⟩]
- ⟨table list⟩ ::= ⟨table spec⟩ [{, ⟨table spec⟩}· · ·]

- ⟨table spec⟩ ::= ⟨nested query expression⟩ [AS ⟨column name⟩]
- ⟨search condition⟩ ::= ⟨boolean term⟩ | ⟨search condition⟩ OR ⟨boolean term⟩
- ⟨boolean term⟩ ::= ⟨boolean factor⟩ | ⟨boolean term⟩ AND ⟨boolean factor⟩
- ⟨boolean factor⟩ ::= [NOT] ⟨boolean primary⟩
- ⟨boolean primary⟩ ::= ⟨predicate⟩ | (⟨search condition⟩)
- ⟨predicate⟩ ::= ⟨comparison predicate⟩ | ⟨between predicate⟩ | ⟨in predicate⟩ | ⟨like predicate⟩
  | ⟨exists predicate⟩ | ⟨null predicate⟩
- ⟨comparison predicate⟩ ::= ⟨value expression⟩ ⟨comp op⟩ ⟨value expression⟩
- ⟨comp op⟩ ::= = | < | > | <= | >= | != | [NOT] ELEMENT OF | [NOT] CONTAINS
  | [NOT] SUBSET OF
- ⟨between predicate⟩ ::= ⟨value expression⟩ [NOT] BETWEEN ⟨value expression⟩ AND
  ⟨value expression⟩
- ⟨in predicate⟩ ::= ⟨value expression tuple list⟩ IN ⟨nested query expression⟩
- ⟨value expression tuple list⟩ ::= ⟨value expression⟩
  | < ⟨value expression⟩ [{, ⟨value expression⟩} ⋯]>
- ⟨like predicate⟩ ::= ⟨not further defined⟩
- ⟨exists predicate⟩ ::= EXISTS ⟨nested query expression⟩
- ⟨null predicate⟩ ::= ⟨column spec⟩ IS [NOT] NULL
- ⟨nested query expression⟩ ::= ⟨table name⟩ | (⟨query expression⟩)
- ⟨column list⟩ ::= [ALL BUT] ⟨column spec⟩ [{, ⟨column spec⟩} ⋯]
- ⟨function⟩ ::= MAX | MIN | AVG | SUM | COUNT | DISTINCT | SUBSUME
- ⟨set operator⟩ ::= UNION | DIFFERENCE | INTERSECT

- ⟨value expression⟩ ::= ⟨term⟩ | ⟨value expression⟩ {+ | -} ⟨term⟩
- ⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩ {* | /} ⟨factor⟩
- ⟨factor⟩ ::= [+ | -] ⟨primary⟩
- ⟨primary⟩ ::= (⟨query expression⟩) | ⟨value spec⟩ | ⟨column spec⟩ | (⟨value expression⟩)

  | ⟨function⟩ (⟨query expression⟩)
- ⟨value spec⟩ ::= ⟨literal⟩ | NULL | (⟨tuple literal⟩)
- ⟨literal⟩ ::= ⟨character string literal⟩ | ⟨numeric literal⟩ | ⟨tuple literal⟩ | ⟨don't care literal⟩
- ⟨tuple literal⟩ ::= < ⟨value spec⟩ [{, ⟨value spec⟩} ···] >
- ⟨column spec⟩ ::= [{⟨reference name⟩.} ···]⟨column name⟩

C.2 DML

- ⟨dml statement⟩ ::= ⟨store statement⟩ | ⟨modify statement⟩ | ⟨erase statement⟩
- ⟨store statement⟩ ::= STORE ⟨table name⟩ [(⟨attribute list⟩)] {VALUES ⟨tuple literal⟩

  | ⟨table name⟩ | ⟨query spec⟩ | ⟨structured query⟩ | ⟨function⟩ (⟨query expression⟩)

  | ⟨query expression⟩ ⟨set operator⟩ ⟨query expression⟩ }
- ⟨attribute list⟩ ::= ⟨column name spec⟩ [{,⟨colun.n name spec⟩} ···]
- ⟨column name spec⟩ ::= column name | column name (⟨ attribute list ⟩)
- ⟨modify statement⟩::= MODIFY ⟨table name⟩ [AS ⟨reference name⟩] SET ⟨set clause⟩ ·

  [WHERE ⟨search condition⟩]
- ⟨set clause⟩ ::= ⟨column name⟩ = {⟨value expression⟩ | (⟨dml statement⟩)}
- ⟨erase statement⟩ ..= ERASE ⟨table name⟩ [AS ⟨reference name⟩][WHERE ⟨search condition⟩]

C.3 DDL

- ⟨ddl statement⟩ ::= ⟨schema⟩ | ⟨scheme⟩ | ⟨drop statement⟩
- ⟨schema⟩ ::= CREATE {⟨table definition⟩ | ⟨view definition⟩}
- ⟨table definition⟩ ::= TABLE ⟨table name⟩ {⟨table element⟩ | ⟨scheme name⟩}
- ⟨table element⟩ ::= (⟨column specification list⟩) | CONSTRAINTS ⟨table constraint definition⟩
- ⟨column specification list⟩ ::= ⟨column specification⟩ [{, ⟨column specification⟩}···]
- ⟨column specification⟩ ::= ⟨column definition⟩ | (⟨table definition⟩)
- ⟨column definition⟩ ::= ⟨column name⟩ ⟨data type⟩ [⟨column constraint spec⟩···]

  [⟨default clause⟩]
- ⟨data type⟩ ::= ⟨character string type⟩ | ⟨numeric type⟩
- ⟨column constraint spec⟩ ::= ⟨not null clause⟩ | ⟨unique clause⟩ | ⟨references clause⟩

  | ⟨check clause⟩
- ⟨not null clause⟩ ::= NOT NULL
- ⟨unique clause⟩ ::= UNIQUE
- ⟨references clause⟩ ::= REFERENCES ⟨column spec⟩ [⟨update rule⟩][⟨delete rule⟩]
- ⟨check clause⟩ ::= CHECK ⟨search condition⟩
- ⟨default clause⟩ ::= DEFAULT ⟨literal⟩
- ⟨table constraint definition⟩ ::= ⟨unique constraint definition⟩ | ⟨referential constraint definition⟩

  | ⟨check constraint definition⟩
- ⟨unique constraint definition⟩ ::= UNIQUE ⟨column list⟩
- ⟨referential constraint definition⟩ ::= REFERENCES ⟨column list⟩ WITH ⟨column list⟩

  [⟨update rule⟩][⟨delete rule⟩]

- ⟨check constraint definition⟩ ::= ⟨check clause⟩ [⟨defer clause⟩]
- ⟨update rule⟩ ::= ⟨action⟩ MODIFY
- ⟨delete rule⟩ ::= ⟨action⟩ ERASE
- ⟨action⟩ ::= CASCADE | NULLIFY | RESTRICT
- ⟨defer clause⟩ ::= IMMEDIATE | DEFFERED
- ⟨view definition⟩ ::= VIEW ⟨table name⟩ AS ⟨query expression⟩
- ⟨scheme⟩ ::= CREATE TYPE ⟨scheme definition⟩
- ⟨scheme definition⟩ ::= ⟨scheme name⟩ ⟨table element⟩
- ⟨drop statement⟩ ::= DROP TABLE ⟨table name⟩ | DROP TYPE ⟨scheme name⟩

  | DROP VIEW ⟨table name⟩

Appendix D. *SQL/NF test cases*

D.1 Data Dictionary

1. QUERY STATEMENT FOLLOWS:

```
CHECK TABLE;


        Table Of Relations
        ******************

Rel Index    Rel Name       Rel Type Index
---------    --------       --------------
0            NEWEMP         10
1            DEPT           16
2            COMPANY        33
3            VIEWD          44
4            EMP            10
5            COMPANIES      20
6            SHORT_EMP      49
7            REL1           50
8            DOMP_V         48
9            NEV            71
10           DEPT_N         75
```

2. QUERY STATEMENT FOLLOWS:

```
CHECK TABLE NEWEMP;

                         Dump of Symbol Table for TABLE: NEWEMP
                         ********************************

INDEX  NAME              LEVEL    DOMAIN         NUMB  PARENT  NEST INDEX
-----  ----              ----     ------         ----  ------  ----------
10     EMP               SCHEME   ON_THE_FLY        5      -1          -2
11     NAME              ATTR     CHAR             32      10          -2
12     AGE               ATTR     INT               1      10          -2
13     DNO               ATTR     INT               2      10          -2
14     CHILDREN          ATTR     PREV_DEFINED      3      10           3
3      CHILD             SCHEME   ON_THE_FLY        3      -1          -2
4      NAME              ATTR     CHAR             32       3          -2
5      AGE               ATTR     INT               1       3          -2
6      TOYS              ATTR     PREV_DEFINED      2       3           0
0      TOY               SCHEME   ON_THE_FLY        2      -1          -2
1      NAME              ATTR     CHAR             32       0          -2
2      COLOR             ATTR     CHAR             32       0          -2
15     PROJECTS          ATTR     PREV_DEFINED      2      10           7
7      PROJECT           SCHEME   ON_THE_FLY        2      -1          -2
8      NAME              ATTR     CHAR             32       7          -2
9      NUMBER            ATTR     INT               2       7          -2
```

3. QUERY STATEMENT FOLLOWS:

```
CHECK TABLE EMP;

                         Dump of Symbol Table for TABLE: EMP
                         ********************************

INDEX  NAME              LEVEL    DOMAIN         NUMB  PARENT  NEST INDEX
-----  ----              ----     ------         ----  ------  ----------
84     EMP2              SCHEME   ON_THE_FLY        4      -1          -2
85     ENO               ATTR     INT               1      84          -2
86     ENAME             ATTR     CHAR             32      84          -2
87     DNO               ATTR     INT               1      84          -2
88     SAL               ATTR     FLOAT             4      84          -2
```

D.2 *SFW-expressions*

1. QUERY STATEMENT FOLLOWS:

```
SELECT ENO,DNAME
FROM DEPT,EMP
WHERE ENAME = "SMITH";



QUERY node follows:
Query node Operator is: PROJECT
Arg name =
Arg reltype =
ARG list follows:
Attrdesc name = DNAME
Attrdesc type = CHAR
Attrdesc size = 32
Attrdesc parentrel = DEPT
Next list follows:
Attrdesc name = ENO
Attrdesc type = INT
Attrdesc size = 1
Attrdesc parentrel = EMP
Left input node follows:

QUERY node follows:
Query node Operator is: CARTESIAN PRODUCT
Arg name =
Arg reltype =
Left input node follows:

QUERY node follows:
Query node Operator is: SELECT
Arg name = DEPT
Arg reltype = DEPARTMENT
Right input node follows:

QUERY node follows:
Query node Operator is: SELECT
Arg name = EMP
Arg reltype = EMP2
Arg pred follows:
Pred oper is =
Pred constant_on_right = TRUE
LEFT pred operand follows:
Op_type is CHAR: ENAME
RIGHT pred operand follows:
Op_type is CHAR: "SMITH"
```

2. QUERY STATEMENT FOLLOWS:

```
SELECT ENAME
FROM (SELECT ALL
      FROM EMP,DEPT
      WHERE DNO < ENO)
WHERE DNAME = "SHIPPING";


QUERY node follows:
Query node Operator is: PROJECT
Arg name =
Arg reltype =
ARG list follows:
Attrdesc name = ENAME
Attrdesc type = CHAR
Attrdesc size = 32
Attrdesc parentrel = EMP
Left input node follows:

QUERY node follows:
Query node Operator is: CARTESIAN PRODUCT
Arg name =
Arg reltype =
Left input node follows:

QUERY node follows:
Query node Operator is: SELECT
Arg name = EMP
Arg reltype = EMP2
Arg pred follows:
Pred oper is <
Pred constant_on_right = FALSE
LEFT pred operand follows:
Op_type is CHAR: DNO
RIGHT pred operand follows:
Op_type is CHAR: ENO
Right input node follows:

QUERY node follows:
Query node Operator is: SELECT
Arg name = DEPT
Arg reltype = DEPARTMENT
Arg pred follows:
Pred oper is =
Pred constant_on_right = TRUE
LEFT pred operand follows:
Op_type is CHAR: DNAME
RIGHT pr    perand follows:
Op_type is  IAR: "SHIPPING"
```

3. QUERY STATEMENT FOLLOWS:

```
SELECT ALL
FROM DEPT,NEWEMP,COMPANY
WHERE DEPT.DNO = NEWEMP.DNO
AND DEPT.LOC = COMPANY.LOC
AND AGE>35;


QUERY node follows:
Query node Operator is: CARTESIAN PRODUCT
Arg name =
Arg reltype =
Arg pred follows:
Pred oper is AND
Pred constant_on_right = FALSE
LEFT pred operand follows:
Op_type is PRED:
Pred oper is =
Pred constant_on_right = FALSE
LEFT pred operand follows:
Op_type is CHAR: DNO
With reference name DEPT
RIGHT pred operand follows:
Op_type is CHAR: DNO
With reference name NEWEMP
RIGHT pred operand follows:
Op_type is PRED:
Pred oper is =
Pred constant_on_right = FALSE
LEFT pred operand follows:
Op_type is CHAR: LOC
With reference name DEPT
RIGHT pred operand follows:
Op_type is CHAR: LOC
With reference name COMPANY
Left input node follows:

QUERY node follows:
Query node Operator is: SELECT
Arg name = DEPT
Arg reltype = DEPARTMENT
Right input node follows:

QUERY node follows:
Query node Operator is: CARTESIAN PRODUCT
Arg name =
Arg reltype =
Left input node follows:

QUERY node follows:
```

```
Query node Operator is: SELECT
Arg name = NEWEMP
Arg reltype = EMP
Arg pred follows:
Pred oper is >
Pred constant_on_right = TRUE
LEFT pred operand follows:
Op_type is CHAR: AGE
RIGHT pred operand follows:
Op_type is INT: 35
Right input node follows:

QUERY node follows:
Query node Operator is: SELECT
Arg name = COMPANY
Arg reltype = COMP
```

4. QUERY STATEMENT FOLLOWS:

```
SELECT DNAME,(SELECT ENAME
              FROM EMP
              WHERE SAL>10000)
FROM COMPANY
WHERE LOC = "CHICAGO";
```

```
QUERY node follows:
Query node Operator is: PROJECT
Arg name =
Arg reltype =
ARG list follows:
Attrdesc name = DNAME
Attrdesc type = CHAR
Attrdesc size = 20
Attrdesc parentrel = COMPANY
Next list follows:
Attrdesc name = EMP
Attrdesc type = PREV_DEFINED
Attrdesc rvatype = EMPs
Attrdesc size = 4
Attrdesc parentrel = COMPANY
Sublist follows:
Attrdesc name = ENAME
Attrdesc type = CHAR
Attrdesc size = 24
Attrdesc parentrel = EMP
Left input node follows:

QUERY node follows:
Query node Operator is: SELECT
Arg name = COMPANY
Arg reltype = COMP
Arg pred follows:
Pred oper is =
Pred constant_on_right = TRUE
LEFT pred operand follows:
Op_type is CHAR: LOC
RIGHT pred operand follows:
Op_type is CHAR: "CHICAGO"
ARG list follows:
Attrdesc name = EMP
Attrdesc type = PREV_DEFINED
Attrdesc rvatype = EMPs
Attrdesc size = 4
Attrdesc parentrel = COMPANY
List cond follows:
Pred oper is >
Pred constant_on_right = TRUE
LEFT pred operand follows:
Op_type is CHAR: SAL
RIGHT pred operand follows:
Op_type is INT: 10000
```

5. QUERY STATEMENT FOLLOWS:

```
SELECT DNAME,DNO,(SELECT ENAME,ENO,(SELECT CNAME
                                    FROM CHILDREN)
                  FROM EMP
                  WHERE SAL >1000)
FROM COMPANY
WHERE LOC = "CHICAGO" AND DNO = 456;
```

```
QUERY node follows:
Query node Operator is: PROJECT
Arg name =
Arg reltype =
ARG list follows:
Attrdesc name = DNAME
Attrdesc type = CHAR
Attrdesc size = 20
Attrdesc parentrel = COMPANY
Next list follows:
Attrdesc name = DNO
Attrdesc type = INT
Attrdesc size = 2
Attrdesc parentrel = COMPANY
Next list follows:
Attrdesc name = EMP
Attrdesc type = PREV_DEFINED
Attrdesc rvatype = EMPs
Attrdesc size = 4
Attrdesc parentrel = COMPANY
Sublist follows:
Attrdesc name = ENAME
Attrdesc type = CHAR
Attrdesc size = 24
Attrdesc parentrel = EMP
Next list follows:
Attrdesc name = ENO
Attrdesc type = INT
Attrdesc size = 2
Attrdesc parentrel = EMP
Next list follows:
Attrdesc name = CHILDREN
Attrdesc type = PREV_DEFINED
Attrdesc rvatype = CHILDRENs
Attrdesc size = 2
Attrdesc parentrel = EMP
Sublist follows:
Attrdesc name = CNAME
Attrdesc type = CHAR
Attrdesc size = 12
Attrdesc parentrel = CHILDREN
Left input node follows:

QUERY node follows:
Query node Operator is: SELECT
Arg name = COMPANY
Arg reltype = COMP
Arg pred follows:
Pred oper is AND
Pred constant_on_right = FALSE
LEFT pred operand follows:
Op_type is PRED:
Pred oper is =
Pred constant_on_right = TRUE
LEFT pred operand follows:
Op_type is CHAR: LOC
RIGHT pred operand follows:
Op_type is CHAR: "CHICAGO"
RIGHT pred operand follows:
Op_type is PRED:
Pred oper is =
Pred constant_on_right = TRUE
LEFT pred operand follows:
Op_type is CHAR: DNO
RIGHT pred operand follows:
Op_type is INT: 456
ARG list follows:
Attrdesc name = EMP
Attrdesc type = PREV_DEFINED
Attrdesc rvatype = EMPs
Attrdesc size = 4
Attrdesc parentrel = COMPANY
List cond follows:
Pred oper is >
Pred constant_on_right = TRUE
LEFT pred operand follows:
Op_type is CHAR: SAL
RIGHT pred operand follows:
Op_type is INT: 1000
```

6. QUERY STATEMENT FOLLOWS:

```
SELECT NAME,DNAME,(SELECT NAME
                   FROM   CHILDREN
                   WHERE  AGE>5)
FROM DEPT,NEWEMP
WHERE DEPT.DNO = NEWEMP.DNO
AND LOC = "DAYTON";


QUERY node follows:
Query node Operator is: PROJECT
Arg name =
Arg reltype =
ARG list follows:
Attrdesc name = DNAME
Attrdesc type = CHAR
Attrdesc size = 32
Attrdesc parentrel = DEPT
Next list follows:
Attrdesc name = NAME
Attrdesc type = CHAR
Attrdesc size = 32
Attrdesc parentrel = NEWEMP
Next list follows:
Attrdesc name = CHILDREN
Attrdesc type = PREV_DEFINED
Attrdesc rvatype = CHILD
Attrdesc size = 3
Attrdesc parentrel = NEWEMP
Sublist follows:
Attrdesc name = NAME
Attrdesc type = CHAR
Attrdesc size = 32
Attrdesc parentrel = CHILDREN
Left input node follows:

QUERY node follows:
Query node Operator is: CARTESIAN PRODUCT
```

```
Arg name =
Arg reltype =
Arg pred follows:
Pred oper is =
Pred constant_on_right = FALSE
LEFT pred operand follows:
Op_type is CHAR: DNO
With reference name DEPT
RIGHT pred operand follows:
Op_type is CHAR: DNO
With reference name NEWEMP
Left input node follows:
QUERY node follows:
Query node Operator is: SELECT
Arg name = DEPT
Arg reltype = DEPARTMENT
Arg pred follows:
Pred oper is =
Pred constant_on_right = TRUE
LEFT pred operand follows:
Op_type is CHAR: LOC
RIGHT pred operand follows:
Op_type is CHAR: "DAYTON"
Right input node follows:

QUERY node follows:
Query node Operator is: SELECT
Arg name = NEWEMP
Arg reltype = EMP
ARG list follows:
Attrdesc name = CHILDREN
Attrdesc type = PREV_DEFINED
Attrdesc rvatype = CHILD
Attrdesc size = 3
Attrdesc parentrel = NEWEMP
List cond follows:
Pred oper is >
Pred constant_on_right = TRUE
LEFT pred operand follows:
Op_type is CHAR: AGE
RIGHT pred operand follows:
Op_type is INT: 5
```

D.3 NEST and UNNEST

1. QUERY STATEMENT FOLLOWS:

```
SELECT (SELECT ALL
        FROM DEPT
        WHERE DNAME = "SMITH"), EMP
FROM (NEST DEPT_N
      ON DNAME AS DEPT)
WHERE DNO = 123;


QUERY node follows:
Query node Operator is: PROJECT
Arg name =
Arg reltype =
ARG list follows:
Attrdesc name = DEPT
Attrdesc type = NEST
Attrdesc rvatype =
Attrdesc size = 1
Attrdesc parentrel = DEPT_N
Next list follows:
Attrdesc name = EMP
Attrdesc type = PREV_DEFINED
Attrdesc rvatype = EMPs
Attrdesc size = 5
Attrdesc parentrel = DEPT_N
Left input node follows:

QUERY node follows:
Query node Operator is: SELECT
Arg name =
Arg reltype =
Arg pred follows:
Pred oper is =
Pred constant_on_right = TRUE
LEFT pred operand follows:
Op_type is CHAR: DNO
RIGHT pred operand follows:
Op_type is INT: 123
ARG list follows:
Attrdesc name = DEPT
Attrdesc type = NEST
Attrdesc rvatype =
Attrdesc size = 1
Attrdesc parentrel = DEPT_N
List cond follows:
Pred oper is =
Pred constant_on_right = TRUE
LEFT pred operand follows:
```

```
Op_type is CHAR: DNAME
RIGHT pred operand follows:
Op_type is CHAR: "SMITH"
Left input node follows:

QUERY node follows:
Query node Operator is: NEST
Arg name = DEPT_N
Arg reltype =
ARG list follows:
Attrdesc name = DNAME
Attrdesc type = CHAR
Attrdesc size = 20
Attrdesc parentrel = DEPT_N
ARG nest name = DEPT
```

2. QUERY STATEMENT FOLLOWS:

```
SELECT DNO,(SELECT ENAME,(SELECT CNAME
                          FROM CHILDREN)
            FROM (NEST EMP
                  ON CNAME,C_DOB AS CHILDREN)
            WHERE SAL > 1500 )
FROM DEPT_N;


QUERY node follows:
Query node Operator is: PROJECT
Arg name =
Arg reltype =
ARG list follows:
Attrdesc name = DNO
Attrdesc type = INT
Attrdesc size = 1
Attrdesc parentrel = DEPT_N
Next list follows:
Attrdesc name = EMP
Attrdesc type = PREV_DEFINED
Attrdesc rvatype = EMPs
Attrdesc size = 5
Attrdesc parentrel = DEPT_N
Sublist follows:
Attrdesc name = ENAME
Attrdesc type = CHAR
Attrdesc size = 24
Attrdesc parentrel = EMP
Next list follows:
Attrdesc name = CHILDREN
Attrdesc type = NEST
Attrdesc rvatype =
Attrdesc size = 2
Attrdesc parentrel = EMP
Sublist follows:
Attrdesc name = CNAME
Attrdesc type = CHAR
Attrdesc size = 12
Attrdesc parentrel = CHILDREN
Left input node follows:

QUERY node follows:
Query node Operator is: NEST
Arg name =
Arg reltype =
ARG list follows:
Attrdesc name = EMP
Attrdesc type = PREV_DEFINED
Attrdesc rvatype = EMPs
Attrdesc size = 5
Attrdesc parentrel = DEPT_N
Sublist follows:
Attrdesc name = CNAME
Attrdesc type = CHAR
Attrdesc size = 12
Attrdesc parentrel = EMP
Next list follows:
Attrdesc name = C_DOB
Attrdesc type = INT
Attrdesc size = 2
Attrdesc parentrel = EMP
ARG nest name = CHILDREN
Left input node follows:

QUERY node follows:
Query node Operator is: SELECT
Arg name = DEPT_N
Arg reltype = DEPT_N
ARG list follows:
Attrdesc name = EMP
Attrdesc type = PREV_DEFINED
Attrdesc rvatype = EMPs
Attrdesc size = 5
Attrdesc parentrel = DEPT_N
List cond follows:
Pred oper is >
Pred constant_on_right = TRUE
LEFT pred operand follows:
Op_type is CHAR: SAL
RIGHT pred operand follows:
Op_type is INT: 1500
```

3. QUERY STATEMENT FOLLOWS:

```
UNNEST NEWEMP ON CHILDREN,PROJECTS;


QUERY node follows:
Query node Operator is: UNNEST
Arg name = NEWEMP
Arg reltype = EMP
ARG list follows:
Attrdesc name = CHILDREN
Attrdesc type = PREV_DEFINED
Attrdesc rvatype = CHILD
Attrdesc size = 3
Attrdesc parentrel = NEWEMP
Next list follows:
Attrdesc name = PROJECTS
Attrdesc type = PREV_DEFINED
Attrdesc rvatype = PROJECT
Attrdesc size = 2
Attrdesc parentrel = NEWEMP
```

4. QUERY STATEMENT FOLLOWS:

```
SELECT ENO,ENAME,DNAME
FROM (UNNEST COMPANY
      ON EMP);


QUERY node follows:
Query node Operator is: PROJECT
Arg name =
Arg reltype =
ARG list follows:
Attrdesc name = DNAME
Attrdesc type = CHAR
Attrdesc size = 20
Attrdesc parentrel = COMPANY
Next list follows:
Attrdesc name = ENO
Attrdesc type = INT
Attrdesc size = 2
Attrdesc parentrel = EMP
Next list follows:
Attrdesc name = ENAME
Attrdesc type = CHAR
Attrdesc size = 24
Attrdesc parentrel = EMP
Left input node follows:

QUERY node follows:
Query node Operator is: UNNEST
```

```
Arg name = COMPANY
Arg reltype = COMP
ARG list follows:
Attrdesc name = EMP
Attrdesc type = PREV_DEFINED
Attrdesc rvatype = EMPs
Attrdesc size = 4
Attrdesc parentrel = COMPANY
```

5. QUERY STATEMENT FOLLOWS:

```
SELECT ENO,ENAME,DNAME
FROM (UNNEST COMPANY
      ON EMP)
WHERE SAL > 1500;


QUERY node follows:
Query node Operator is: PROJECT
Arg name =
Arg reltype =
ARG list follows:
Attrdesc name = DNAME
Attrdesc type = CHAR
Attrdesc size = 20
Attrdesc parentrel = COMPANY
Next list follows:
Attrdesc name = ENO
Attrdesc type = INT
Attrdesc size = 2
Attrdesc parentrel = EMP
Next list follows:
Attrdesc name = ENAME
Attrdesc type = CHAR
Attrdesc size = 24
Attrdesc parentrel = EMP
Left input node follows:

QUERY node follows:
Query node Operator is: SELECT
Arg name =
Arg reltype =
Arg pred follows:
Pred oper is >
Pred constant_on_right = TRUE
LEFT pred operand follows:
Op_type is CHAR: SAL
RIGHT pred operand follows:
Op_type is INT: 1500
Left input node follows:
```

```
QUERY node follows:
Query node Operator is: UNNEST
Arg name = COMPANY
Arg reltype = COMP
ARG list follows:
Attrdesc name = EMP
Attrdesc type = PREV_DEFINED
Attrdesc rvatype = EMPs
Attrdesc size = 4
Attrdesc parentrel = COMPANY
```

6. QUERY STATEMENT FOLLOWS:

```
SELECT NEWEMP.NAME,CHILDREN.NAME
FROM (UNNEST NEWEMP
      ON CHILDREN);
```

```
QUERY node follows:
Query node Operator is: PROJECT
Arg name =
Arg reltype =
ARG list follows:
Attrdesc name = NAME
Attrdesc type = CHAR
Attrdesc size = 32
Attrdesc parentrel = NEWEMP
Next list follows:
Attrdesc name = NAME
Attrdesc type = CHAR
Attrdesc size = 32
Attrdesc parentrel = CHILDREN
Left input node follows:

QUERY node follows:
Query node Operator is: UNNEST
Arg name = NEWEMP
Arg reltype = EMP
ARG list follows:
Attrdesc name = CHILDREN
Attrdesc type = PREV_DEFINED
Attrdesc rvatype = CHILD
Attrdesc size = 3
Attrdesc parentrel = NEWEMP
```

Bibliography

1. American National Standards Commitee on Computer and Information Processing. "Relational Database language (X3H2-84-2)." Draft Proposed, January 1984.
2. Carey, Michael J. and others. "The EXODUS Extensible DBMS Project: An Overview." In Zdonik, Stanley B. and David Maier, editors, *Readings in Objected-Oriented Database Systems*, San Mateo, California: Morgan Kaufman Publishers, Inc, 1990.
3. Codd, E.F "A Relational Model for Large Shared Data Banks," *Communications ACM*, *6*(13):377–387 (June 1970).
4. Colby, Latha S. *A Recursive Algebra for Nested Relations*. Computer Sciences Technical Report 259, Indiana University, January 1989.
5. Deshpande, V. and P.A. Larson. *An Algebra for Nested Relations*. Technical Report CS-87-65, University of Waterloo, 1987.
6. Herjee, K.B. and R. Sadeghi. "Rapid implementation of SQL: a case study using YACC and LEX," *Information and Software Technology*, *30*:228–236 (May 1988).
7. Jaeshke, G. and H.J. Schek. "Remarks on the Algebra on Non-First Normal Form Relations." In *Proc. 1st PODS*, pages 124–138, 1982.
8. Kirkpatrick, James E. "The Natural Join of Nested Relations." Unpublished technical report. Air Force Institute of Technology (AU), Wright-Patterson AFB OH, February 1989.
9. Makinouchi, A. "A Consideration of Normal Form of Not-Necessarily-Normalized Relations in the Relational Data Model." In *Proceedings of 3rd. VLDB*, pages 447–453, 1977.
10. Mankus, Michael A. *Design and Implementation of the Nested Relational Data Model under the EXODUS Extensible Database System*. MS thesis, AFIT/GCS/ENG/89D-11, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989.
11. Ramakrishnan, Srinivasan. *Design and Implementation of a Translator for SQL/NF with Role Joins*. MS thesis, University of Texas at Austin, Austin Texas, December 1986.
12. Richardson, J. and others;. *The Design of the E programming Language*. Technical Report, University of Wisconsin-Madison, February 1989.
13. Roth, Mark A. and others. "SQL/NF: A Query Language for ¬1NF Relational Databases," *Information Systems*, *12*(1):99–114 (January 1987).
14. Schek, H.J. and M.H. Scholl. "The Relational Model with Relation-Valued Attributes," *Information Systems*, *11*(2):137–147 (1986).
15. Stonebraker, Michael and others. "Extending a Database System with Procedures," *ACM Traqnsactions on Database Systems*, *12*(3):350–376 (September 1987).
16. Stroustrup, Bjarne. *The C++ Programming Language*. Reading, Massachusetts. Addison-Wesley Publishing Company, 1986.
17. Thomas, S.J. and P.C. Fischer. "Nested Relational Structures." In Kanellakis, P.C., editor, *Advances in Computing Research III, The theory of Databases*, pages 269–307, JAI Press, 1986.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed and completing and reviewing the collection of information Send comments regarding this burden estimate or any other aspect of this collection of information including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1990	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE SQL/NF Translator for the TRITON Nested Relational Database System	5. FUNDING NUMBERS
6. AUTHOR(S) Craig W. Schnepf, Capt, USAF	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCE/ENG/90D-05
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

[illegible]

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public relaease; distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The problem addressed in this thesis effort concerns the design and implementation of a high level data base query language translator based on the nested relational data model. The objective of the model is to increase the performance of the relational model by modeling real-world objects in the problem domain into nested relations. The translator is designed within the EXODUS extensible architectural framework for building application-specific database systems. The SQL/NF query language used for the nested relational model is an extension of the popular relational model query language SQL. The query language is translated into a nested relational algebra (Colby algebra) in the form of a query tree structure. A large amount of theory exists for the nested relational model, however, very little information on the implementation of a high level query language for the model is available. This thesis effort provided the front end to a proto-type nested relational data base management system (Triton) using the EXODUS tool kit.

14. SUBJECT TERMS Nested Relational Database Model, SQL/NF, EXODUS, TRITON, Colby Algebra, LEX, YACC	15. NUMBER OF PAGES 121
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to **stay within the lines to meet optical scanning requirements.**

Block 1. Agency Use Only (Leave Blank)

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Names(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency. Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of ..., To be published in .... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availablity Statement. Denote public availability or limitation. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR)

DOD	- See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE	- See authorities
NASA	- See Handbook NHB 2200.2.
NTIS	- Leave blank.

Block 12b. Distribution Code.

DOD	- DOD - Leave blank
DOE	- DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports
NASA	- NASA - Leave blank
NTIS	- NTIS - Leave blank.

Block 13. Abstract. Include a brief (Maximum 200 words) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (NTIS only).

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.